Analisi statica del codice

Motivazioni

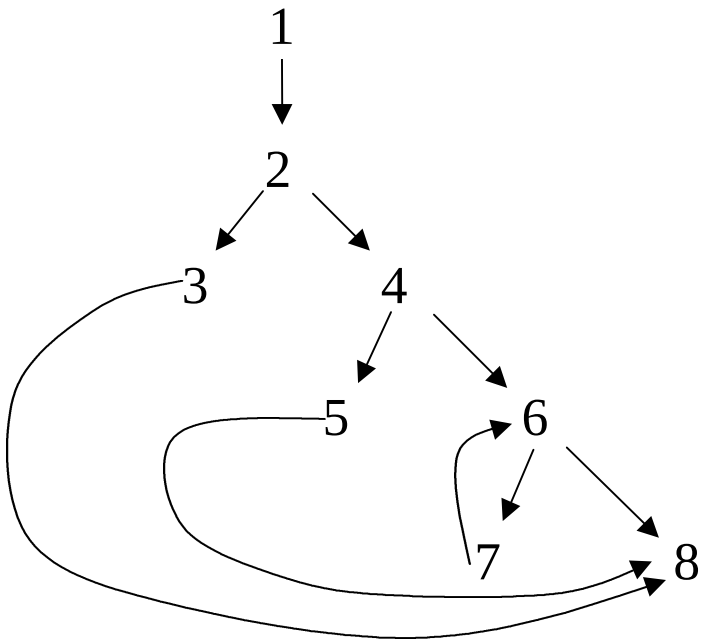
Aiuta gli ingegneri del sw durante la manutenzione fornendo info che facilitano la comprensione del programma e consentono di valutare l'impatto di un cambiamento → supporta

- Testing strutturale
- Reverse engineering
- Reengineering
- Restructuring

Control flow graph (CFG)

CFG	Codice
Nodo n	Istruzione
Nodo iniziale n_1	Istruzione iniziale
Nodo finale n_N (unico)	Istruzione finale (unica, anche fittizia)
Arco orientato fra i nodi n e m	L'istruzione m è sintatticamente uno dei diretti successori dell'istruzione n

```
main()
{
1  scanf("%d", &a);
2  if (a == 3)
3     x = a;
4  else if (a == 4)
5           a++;
6       else while (x)
7           x--;
8  printf("%d %d", a, x);
}
```



Analisi di flusso

Esiste una procedura generale usata per determinare le proprietà di un dato CFG propagando appropriati flussi di info entro di esso; tali flussi cambiano durante la propagazione a seconda della computazione effettuata da ciascuna istruzione del programma

V: insieme dei valori che possono essere propagati

F: insieme delle funzioni di trasferimento f_n: V $\rightarrow$ V, una per ogni nodo n

Operatore di confluenza $\wedge$ (meet)

IN[n] e OUT[n]: insieme dei valori di V in ingresso al nodo n e in uscita dal nodo n, tali che:

$OUT[n] = f_n(IN[n])$

$IN[n] = \wedge_{p \in pred(n)} OUT[p]$ se la propagazione è in avanti (forward)

$IN[n] = \wedge_{s \in succ(n)} OUT[s]$ se la propagazione è all'indietro (backward)

Algoritmo di analisi di flusso

(propagazione in avanti)

1 **for each** node n
2 **do** IN[n] = initial value
3 OUT[n] = f_n(IN[n])

Inizializzazione

4 **while** any OUT[n] changes
5 **do** **for each** node n
6 **do** IN[n] = $\wedge_{p \in pred(n)}$ OUT[p]
7 OUT[n] = f_n(IN[n])

Propagazione

Definizioni di raggiungibilità

Fra i nodi n e m sussiste

- una definizione raggiungente (reaching definition) sulla variabile x se x è definita in n ed esiste un cammino di CFG fra n e m lungo cui x non è ridefinita in alcun nodo distinto sia da n che da m; essa si indica con la tripla $(n,m,x) \in RD$
- un uso raggiungibile (reachable use) sulla variabile x se x è usata in m ed esiste un cammino di CFG fra n e m lungo cui x non è definita in alcun nodo distinto sia da n che da m; esso si indica con la tripla $(n,m,x) \in RU$
- una dipendenza dei dati (data dependence o def-use chain) sulla variabile x se x è definita in n e usata in m ed esiste un cammino di CFG fra n e m lungo cui x non è ridefinita in alcun nodo distinto sia da n che da m; essa si indica con la tripla $(n,m,x) \in DU$

Nota. In tutte e tre le definizioni n e m non sono necessariamente distinti; tuttavia essi possono coincidere solo se l'istruzione corrispondente è ciclica (e quindi il nodo sorgente rappresenta una iterazione precedente rispetto a quella del nodo destinazione)

Calcolo delle definizioni raggiungenti

- $V = \{\{(\nu, x)\} \mid \nu$ è un nodo di CFG che definisce la variabile $x\}$
- $f_n(IN[n]) = GEN_n \cup (IN[n] \setminus KILL_n)$

  dove

  $DEF = \{(\nu, x) \mid \nu$ è un nodo di CFG che definisce la variabile $x\}$
  $GEN_n = \{(n, x) \mid (n, x) \in DEF\}$
  $KILL_n = \{(k, x) \mid (n, x) \in DEF, k$ è un nodo di CFG$\}$
- $\wedge$ è $\cup$
- $\forall$ nodo n di CFG, il valore iniziale di IN[n] è $\varnothing$
- Il verso di propagazione è in avanti

Definizioni raggiungenti: un esempio

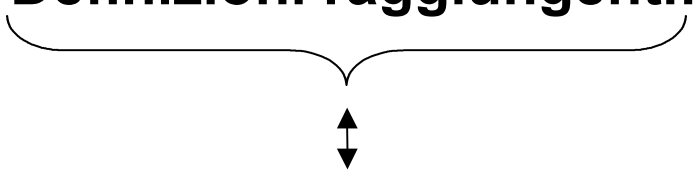
Si adotta la notazione x:n per (n,x); il simbolo - evidenzia l'assenza di cambiamenti

	Inizializzazione		Propagazione	
`main()` `{`	IN	OUT	IN	OUT
`1 scanf("%d", &a);`	∅	{a:1}	-	-
`2 if (a == 3)`	∅	∅	{a:1}	∅∪({a:1}\∅)={a:1}
`3 x = a;`	∅	{x:3}	∅	{x:3}∪∅={x:3}
`4 else if (a == 4)`	∅	∅	∅	∅∪∅=∅
`5 a++;`	∅	{a:5}	∅	{a:5}∪∅={a:5}
`6 else while (x)`	∅	∅	{x:7}	∅∪({x:7}\∅)={x:7}
`7 x--;`	∅	{x:7}	∅	{x:7}∪∅={x:7}
`8 printf("%d %d",a,x);`	∅	∅	{x:3;a:5}	∅∪({x:3;a:5}\∅)= {x:3;a:5} INUTILE
`}`				

Definizioni raggiungenti: un esempio (cont.)

	Propagazione					
	IN	OUT	IN	OUT	IN	OUT
1	-	-	-	-	-	-
2	-	-	-	-	-	-
3	{a:1}	{x:3}∪({a:1}\∅)= {x:3;a:1}	-	-	-	-
4	{a:1}	∅∪({a:1}\∅)= {a:1}	-	-	-	-
5	∅	-	{a:1}	{a:5}	-	-
6	{x:7}	-	{x:7; a:1}	∅∪({x:7;a:1}\∅)= {x:7;a:1}	-	-
7	{x:7}	{x:7}	-	-	{x:7; a:1}	{x:7; a:1}
8	{x:3,7; a:5}	∅∪({x:3,7;a:5}\∅) ={x:3,7;a:5} INUTILE	{x:3,7; a:1,5}	∅∪({x:3,7;a:1,5}\∅) ={x:3,7;a:1,5} INUTILE	{x:3,7; a:1,5}	-

Definizioni raggiungenti: interpretazione dei risultati



	IN	Elementi di RD
1	∅	
2	{a:1}	*(1,2,a)*
3	{a:1}	*(1,3,a)*
4	{a:1}	*(1,4,a)*
5	{a:1}	*(1,5,a)*
6	{x:7;a:1}	(1,6,a) *(7,6,x)*
7	{x:7;a:1}	(1,7,a) *(7,7,x)*
8	{x:3,7;a:1,5}	*(1,8,a) (5,8,a) (3,8,x) (7,8,x)*

nodo finale

nodo iniziale

Nota. In italico sono evidenziati gli elementi appartenenti a DU

Calcolo degli usi raggiungibili

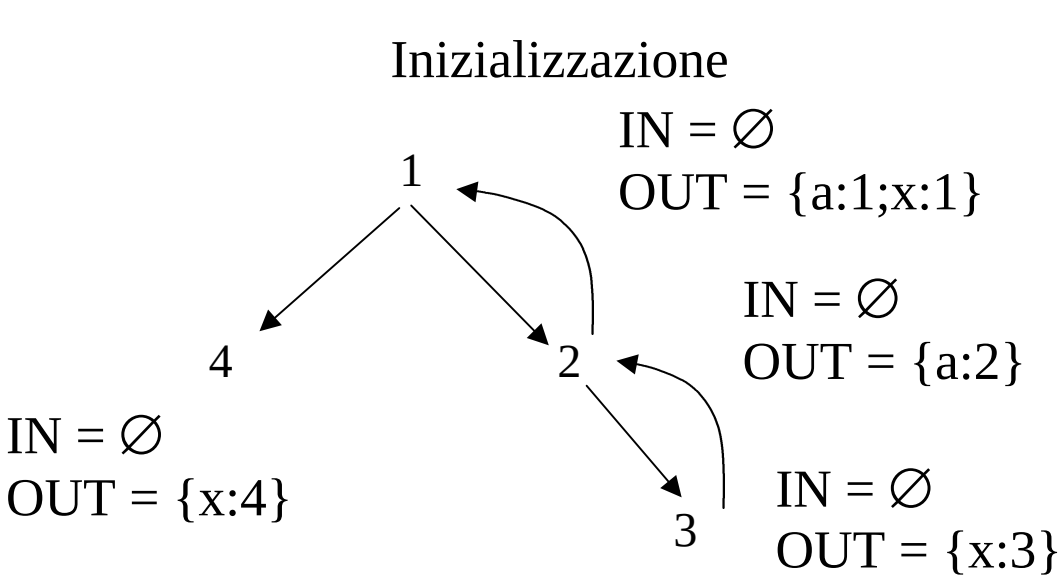
- $V = \{\{(\nu, x)\} \mid \nu$ è un nodo di CFG che usa la variabile $x\}$
- $f_n(IN[n]) = GEN_n \cup (IN[n] \setminus KILL_n)$

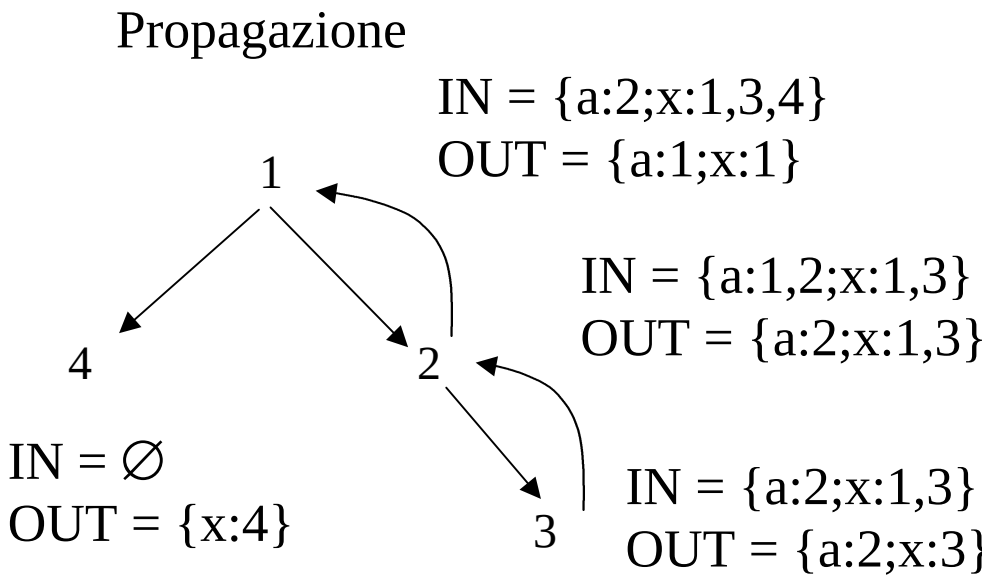
  dove

  $USE = \{(\nu, x) \mid \nu$ è un nodo di CFG che usa la variabile $x\}$
  $GEN_n = \{(n, x) \mid (n, x) \in USE\}$
  $KILL_n = \{(k, x) \mid (n, x) \in DEF, k$ è un nodo di CFG$\}$
- $\wedge$ è $\cup$
- $\forall$ nodo n di CFG, il valore iniziale di IN[n] è $\varnothing$
- Il verso di propagazione è all'indietro

Usi raggiungibili: un esempio

```
main()
{
1 while ((a--)&&(x++))
2      while (a++)
3          x--;
4 a = x + 1;
}
```





Usi raggiungibili: interpretazione dei risultati

	IN	Elementi di RU
1	{a:2;x:1,3,4}	*(1,2,a)* *(1,1,x)* *(1,3,x)* *(1,4,x)*
2	{a:1,2;x:1,3}	*(2,1,a)* *(2,2,a)* (2,1,x) (2,3,x)
3	{a:2;x:1,3}	(3,2,a) *(3,1,x)* *(3,3,x)*
4	∅	

nodo finale

nodo iniziale

Nota. In italico sono evidenziati gli elementi appartenenti a DU

Calcolo delle dipendenze dei dati (cioè dei cammini definizione-uso)

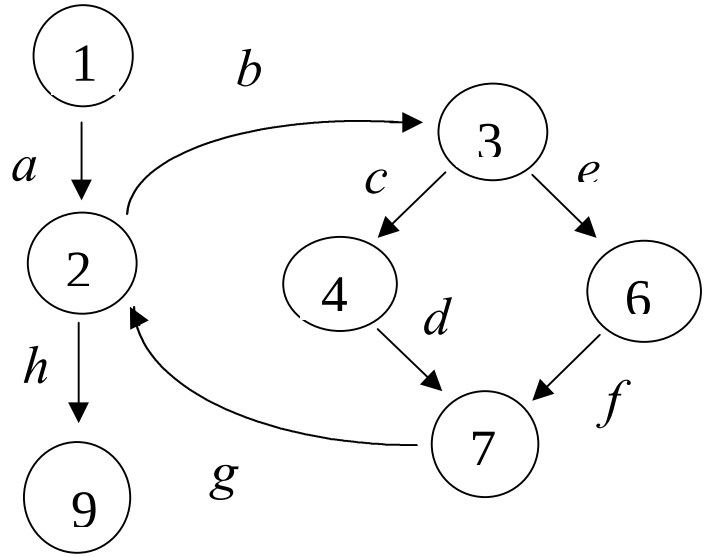
L'insieme delle dipendenze dei dati si può ottenere come

- restrizione dell'insieme delle definizioni raggiungenti a tutte e sole quelle triplette che soddisfano la condizione aggiuntiva che il nodo finale usi la variabile considerata; cioè $(n,m,x) \in DU$ sse $(n,m,x) \in RD \wedge m$ usa x, oppure
- restrizione dell'insieme degli usi raggiungibili a tutte e sole quelle triplette che soddisfano la condizione aggiuntiva che il nodo iniziale definisca la variabile considerata; cioè $(n,m,x) \in DU$ sse $(n,m,x) \in RU \wedge n$ definisce x

Espressione dei cammini (path expression)

Rappresentazione algebrica di tutti i cammini di un grafo

Esempio.



Espressione dei cammini = $a(b(cd + ef)g)^*h$

Algebra dei cammini

Arco (edge) del grafo: è etichettato da un identificatore unico (lettera minuscola, ad es. *a*)

Concatenazione di archi (cammini): è rappresentata attraverso l'operatore prodotto (che non si indica, ad es. *ab*) o attraverso l'operatore potenza se lo stesso arco (cammino) viene ripetuto più volte (cioè un n° di volte pari all'esponente), tornando al nodo di partenza

$a^0 = 1$ è la rappresentazione del cammino ciclico di lunghezza zero (associato al nodo da cui il percorso ciclico si diparte e ritorna)

Alternativa fra archi (cammini): è rappresentata attraverso l'operatore somma (ad es. $a + b$)

0 è la rappresentazione del cammino nullo (associato a un grafo vuoto)

Algebra dei cammini (cont.)

Proprietà associativa

$a(bc) = (ab)c = abc$ $\qquad\qquad$ $a + (b + c) = (a + b) + c = a + b + c$

Proprietà commutativa

$a + b = b + a$

Proprietà distributiva

$a(b + c) = ab + ac$ $\qquad\qquad$ $(a + b)c = ac + bc$

Regola di assorbimento

$a + a = a$

Algebra dei cammini (cont.)

Identità

$$1 + 1 = 1$$
$$1\ a = a\ 1 = a$$
$$1^n = 1$$
$$a + 0 = 0 + a = a$$
$$a\ 0 = 0\ a = 0$$

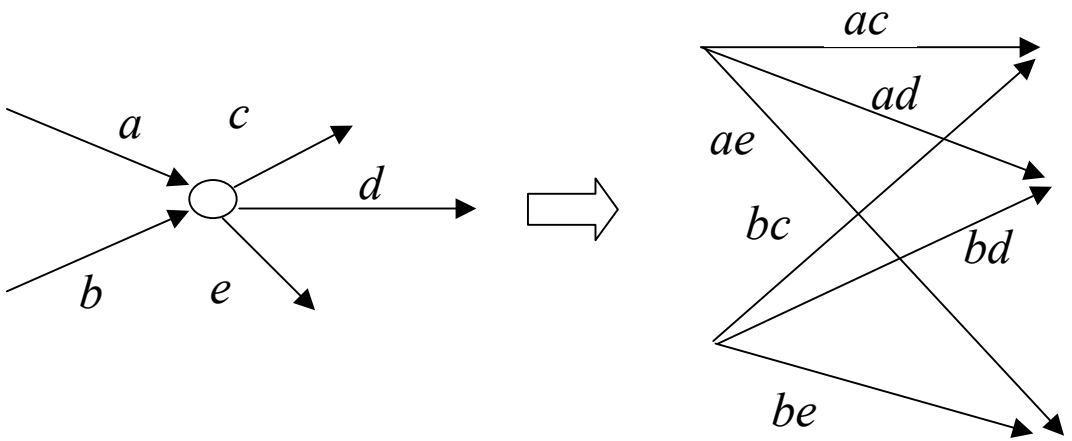
Semplificazione dei cicli

$$a^* = a^0 + a^1 + a^2 + a^3 + \ldots$$
$$aa^* = a^*a = a^+$$

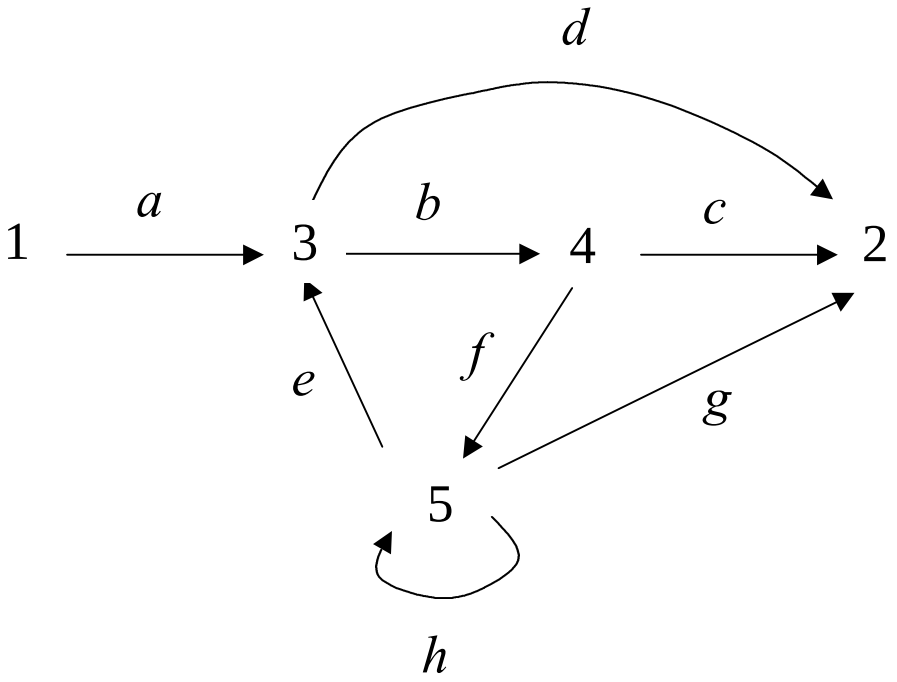
Calcolo dell'espressione dei cammini: algoritmo di riduzione dei nodi

1 rimuovere gli autoanelli X concatenando a X* gli archi uscenti
2 combinare gli archi in serie moltiplicandoli (ed eliminando i nodi intermedi)
3 combinare i percorsi paralleli sommandoli (ed eliminando i nodi intermedi)
4 **while** numero dei cammini (dal nodo iniziale a quello finale) > 1
5 **do** selezionare un nodo k
6 rimuovere gli autoanelli (su k) come in 1
7 applicare a k il passo di cross-term
8 combinare i percorsi in serie come in 2
9 combinare i percorsi paralleli come in 3
10 rimuovere k

Esemplificazione del passo di cross-term



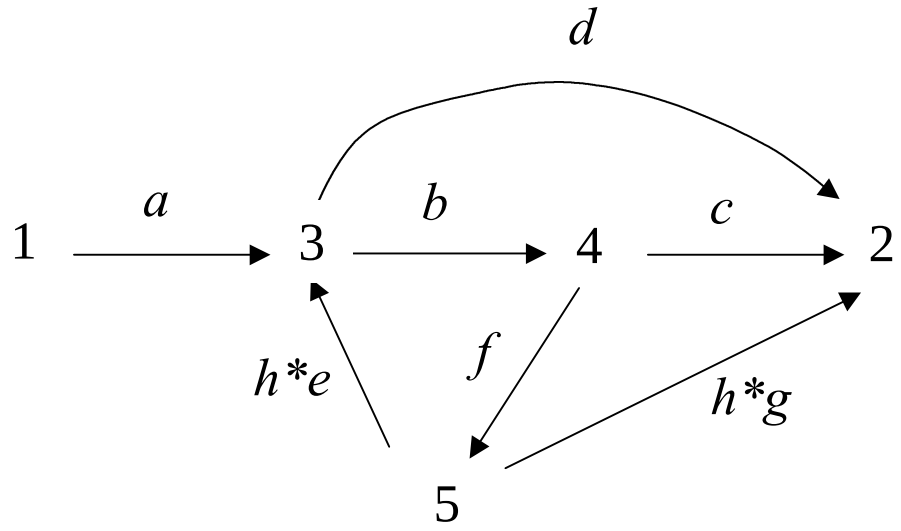
Esempio di applicazione dell'algoritmo di riduzione dei nodi



	1	2	3	4	5
1			*a*		
2					
3		*d*		*b*	
4		*c*			*f*
5		*g*	*e*		*h*

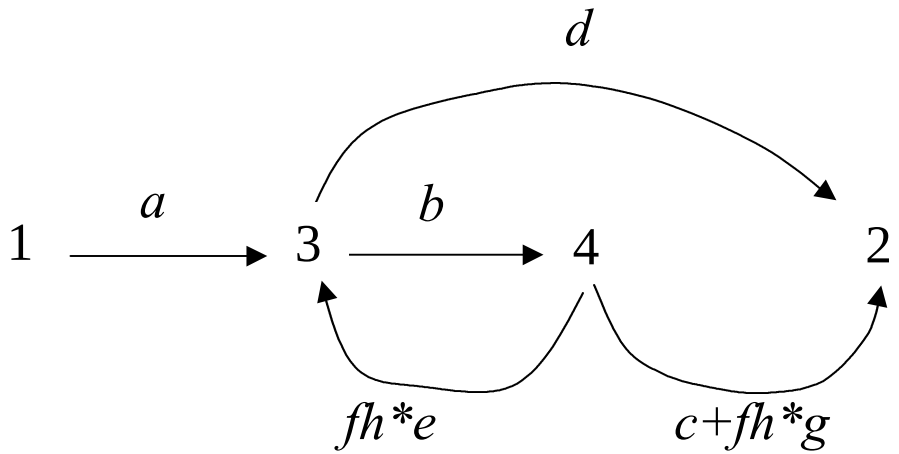
Le caselle vuote della matrice in realtà contengono 0

Prima dell'esecuzione del ciclo



Esempio di applicazione dell'algoritmo di riduzione dei nodi (cont.)

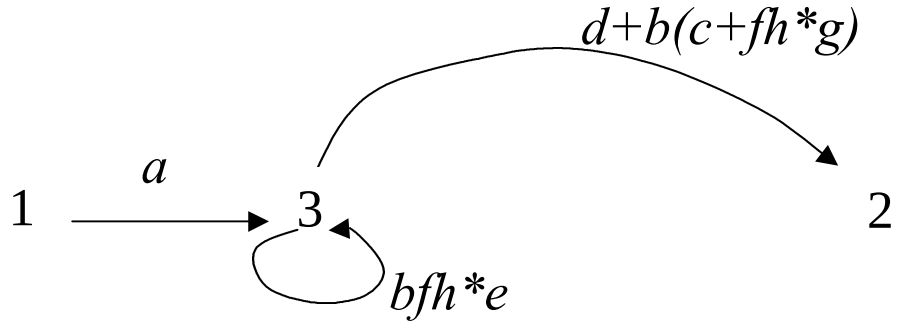
Dopo aver selezionato e rimosso il nodo 5



	1	2	3	4
1			*a*	
2				
3		*d*		*b*
4		*c+fh*g*	*fh*e*	

Matrice ottenuta dopo l'iterazione con r = 5

Dopo aver selezionato e rimosso il nodo 4

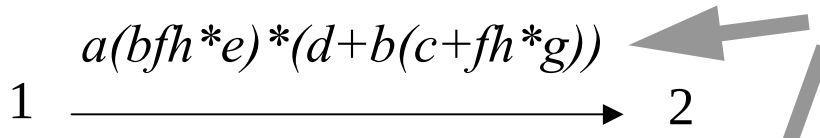


	1	2	3
1			*a*
2			
3		*d+b(c+fh*g)*	*bfh*e*

Matrice ottenuta dopo l'iterazione con r = 4

Esempio di applicazione dell'algoritmo di riduzione dei nodi (cont.)

Dopo aver selezionato e rimosso il nodo 3



espressione dei cammini

	1	2
1		*a(bfh*e)*(d+b(c+fh*g))*
2		

Matrice ottenuta dopo l'iterazione con r = 3

Matrice del grafo

A = [a_{ij}] rappresentazione matriciale di un grafo, dove
a_{ij} = etichetta dell'arco diretto dal nodo n_i al nodo n_j (0 se tale arco $\nexists$)

I loop (elementi della diagonale) possono essere rimossi pre-moltiplicando tutti gli elementi della stessa riga per la potenza *; ad es. se $a_{ii} = h$, si effettuano i seguenti assegnamenti:

$a_{ii} \leftarrow 0$

$a_{ij} \leftarrow h^* a_{ij} \qquad \forall j \neq i$

Algoritmo di riduzione dei nodi operante sulla matrice del grafo

1 costruire una matrice del grafo con le righe 1 e 2
  associate rispettivamente ai nodi di ingresso e uscita
2 **while** n° righe r > 2
3 **do** **if** $a_{rr} = h \neq 0$
4 $a_{rj} \leftarrow h * a_{rj}$ $\forall j \neq r$
5 **for** i ← 1 to r – 1
6 **do for** j ← 1 to r – 1
7 **do** $a_{ij} \leftarrow a_{ij} + a_{ir}\, a_{rj}$
8 ▷ i due cicli for annidati effettuano il passo di cross-term
9 r ← r – 1
10 ▷ cancella la riga r e la colonna r dalla matrice
  (equivale a eliminare un nodo)

Generazione automatica dei casi di test

1) Determinazione, mediante l'algoritmo di riduzione dei nodi, dell'espressione dei cammini del CFG del programma considerato;
2) nota tale espressione, generazione dei cammini del CFG che soddisfano un dato criterio di testing (il criterio di testing specifica come manipolare alternative e cicli);
3) tali cammini diventano casi di test una volta che sono stati determinati, ad es. attraverso una esecuzione simbolica, valori di ingresso corrispondenti

Attenzione: possono esistere cammini impercorribili

Cammini impercorribili (infeasible paths)

Vettore d'ingresso di un cammino (input vector) = sequenza dei valori che il programma acquisisce in ingresso dal mondo esterno lungo il cammino, come fissati prima di iniziare l'esecuzione (costanti, ingressi forniti dall'utente alla tastiera, record di una base di dati, dati di uno o più file, ecc.)

Predicato di un cammino (path predicate o path condition o path expression) = congiunzione (cioè, prodotto logico) delle condizioni che, se vere, consentono di attraversare gli archi del cammino; in tale predicato le variabili dipendenti sono sostituite da espressioni che contengono solo variabili indipendenti, cioè solo elementi del vettore d'ingresso

Cammino impercorribile: cammino per cui non esiste alcun vettore d'ingresso che soddisfi il predicato del cammino stesso

CASE tool di supporto: strumentatori di codice

Determinazione del predicato di un cammino (symbolic execution o predicate interpretation)

Seguire il cammino di interesse; a ogni istruzione:

- se a una variabile x è assegnato un valore d'ingresso, aggiornare l'indicizzazione progressiva di x (solitamente si comincia con x_0) e porre il suo valore simbolico uguale a `*input*`
- se a una variabile x è assegnato il valore di un'espressione, aggiornare l'indicizzazione progressiva di x e porre il suo valore simbolico uguale all'espressione, in cui a tutte le variabili che non sono d'ingresso è sostituito il relativo valore simbolico (con l'indice più recente)
- se si incontra una condizione, congiungerla al predicato del cammino dopo aver sostituito a ciascuna delle variabili che non sono d'ingresso il relativo valore simbolico (con l'indice più recente)

Determinazione del predicato di un cammino: Esempio 1

Problema: determinare se il cammino 1,2,3,4,5,6,7,8 del seguente programma è percorribile o meno, assunto che i valori delle variabile x acquisiti da tastiera siano numeri naturali

```
1  scanf("%d", &x);
2  a = x + 1;
3  b = x - 1;
4  if (a < 10)
5     x++;
6  if (b > 20)
7     x--;
8  printf("%d\n", x);
```

Soluzione
(La sequenza data è effettivamente un cammino solo se è un'istanza dell'espressione dei cammini)

Determinazione del predicato di un cammino: Esempio 1 (cont.)

Sequenza delle istruzioni del cammino	Valori simbolici delle variabili	Condizioni del predicato del cammino
`1 scanf("%d", &x);`	x_0 = `*input*`	
`2 a = x + 1;`	$a_0 = x_0 + 1$	
`3 b = x - 1;`	$b_0 = x_0 - 1$	
`4 if (a < 10) /* con condizione TRUE */`		$a_0 < 10$ che diventa $x_0 + 1 < 10$ cioè $x_0 < 9$
`5 x++;`	$x_1 = x_0 + 1$	
`6 if (b > 20) /* con condizione TRUE */`		$b_0 > 20$ che diventa $x_0 - 1 > 20$ cioè $x_0 > 21$
`7 x--;`	$x_2 = x_0$	
`8 printf("%d\n", x);`		

↓

Predicato del cammino: $x_0 < 9 \wedge x_0 > 21 \rightarrow$ non esiste alcun vettore d'ingresso che soddisfi il predicato $\rightarrow$ il cammino 1,2,3,4,5,6,7,8 è impercorribile

Determinazione del predicato di un cammino: Esempio 2

Problema: determinare se il cammino 1,2,3,4,5,2,3,4,6,2,7 del seguente programma è percorribile o meno, assunto che i valori delle variabile x acquisiti da tastiera siano numeri naturali

```
1  x = 1;
2  while (x > 0)
   {
3     scanf("%d", &x);
4     if (x > 10)
5        x++;
6     else x--;
   }
7  printf("%d\n", x);
```

Soluzione

(La sequenza data è effettivamente un cammino solo se è un'istanza dell'espressione dei cammini)

Determinazione del predicato di un cammino: Esempio 2 (cont.)

Sequenza delle istruzioni del cammino	Valori simbolici delle variabili	Condizioni del predicato del cammino
`1 x = 1;`	$x_0 = 1$	
`2 while (x > 0) /* con condizione TRUE */`		$x_0 > 0$, cioè $1 > 0$ che vale TRUE
`3 scanf("%d", &x);`	$x_1 =$ `*input*`	
`4 if (x > 10) /*con condizione TRUE */`		$x_1 > 10$
`5 x++;`	$x_2 = x_1 + 1$	
`2 while (x > 0) /*con condizione TRUE */`		$x_2 > 0$ che diventa $x_1 + 1 > 0$ cioè $x_1 > -1$
`3 scanf("%d", &x);`	$x_3 =$ `*input*`	
`4 if (x > 10) /*con condizione FALSE */`		$x_3 \leq 10$
`6 else x--;`	$x_4 = x_3 - 1$	
`2 while (x > 0) /*con condizione FALSE */`		$x_4 \leq 0$ che diventa $x_3 - 1 \leq 0$ cioè $x_3 \leq 1$
`7 printf("%d\n", x);`		

↓

Predicato del cammino: $x_1 > 10 \wedge x_3 \leq 1 \rightarrow$ esistono vettori d'ingresso che soddisfano il predicato $\rightarrow$ il cammino 1,2,3,4,5,2,3,4,6,2,7 è percorribile