Program Slicing

Shrawan Kumar

Senior Scientist @ TCS Research

shrawan.kumar@tcs.com

Agenda

- Introduction
- What is program slicing
- Why program slicing
- Variants of program slicing
- Computing a program slice

Introduction

- The size and complexity of a software getting harder to understand, maintain and test

- One might have had questions like
  - Where are the values that flow into this statement, coming from?
  - If I change this statement, what pieces of the program are going to be affected?
  - How can I limit the functionality to only what I need?

Example

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
  s = s + j
  p = p * j
  i = i + 1
  j = j + 1
endwhile
write(s)
write(p)
```

Question:

Which statements impact value of p at last line ?

Example

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
  s = s + j
  p = p * j
  i = i + 1
  j = j + 1
endwhile
write(s)
write(p)
```

Question:

Which statements impact value of p at last line ?

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
  s = s + j
  p = p * j
  i = i + 1
  j = j + 1
endwhile
write(s)
write(p)
```

Example

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

Piece of code (slice) relevant to computing value of p at last line

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

History

- Introduced by Mark D. Weiser (July 23, 1952 – April 27, 1999)
  - As his seminal PhD thesis 1979, published as paper in 1981, 1984
  - Argued that a programmer intuitively tries to slice a program to debug it.
- Since then program slicing has been widely studied and explored with many more variants proposed

What is a program slice?

- Informal: “which statements affect value of variable v at statement *s*”

- Formal:
    - Pair <*s*,v> is called a slicing criterion
    - ”For the slicing criterion <*s*,v> , the slice P’=S(P, <*s*,v>), includes only those statements of P that are needed to capture the behavior of v at *s*.”

- Slicing criterion is generalized to <*s*, X> where X is a set of variables

What is a program slice?

- P' is result of deleting some statements from P, such that P' remains syntactically correct and executable

- P and P' are indistinguishable when their behaviour is observed through "same corresponding window", defined by the slicing criterion

  - For all corresponding executions of P and P' (on same input) :

    Value of X at every occurrence of *s* in execution of P is same as value of X at corresponding occurrence of *s* in execution of P'

What is program slicing ?

- A decomposition technique to extract statements, relevant to a particular computation, from a program

- It describes a mechanism which allows the automatic generation of a slice

Variants of program slicing

- Static
  - Backward
  - Forward
- Dynamic
  - Slice behavior equivalence is only for one execution (input)
- Conditioned
  - A combination

The program slice introduced by Weiser is now called executable backward static slice

Why program slicing ?

- Debugging
- Incremental regression testing
- Program comprehension
- Semantic differences in two versions of program
- A preprocessing step in program verification for scaling up
- …

Program slice definition : issues

- Program P and P’ are indistinguishable when their behaviour is observed through “same corresponding window”, defined by the slicing criterion <*s*, X>

- Statement s may not be there in P’
  - We can observe values in P’ at statement *s’* which is nearest successor to *s*, appearing in P’
- What if P does not terminate

Slicing computation and termination

```
1.  read(x)
2.  i = 1
3.  if (x==0)
4.    while (i<10)
5.      i=i+0
6.    endwhile
7.    x=1
8.  else
9.    x=2
10. endif
11. write(x)
```

```
1.  read(x)
2.  i = 1
3.  if (x==0)
4.    while (i<10)
5.      i=i+1
6.    endwhile
7.    x=1
8.  else
9.    x=2
10. endif
11. write(x)
```

- Slicing criterion <11,{x}>
- First program:
  - For x ==0, no termination (no output)
  - For x!=0, output 2
- Second program:
  - For x ==0, output 1
  - For x!=0, output 2

Slicing computation and termination

```
1.  read(x)
2.  i = 1
3.  if (x==0)
4.    while (i<10)
5.      i=i+0
6.    endwhile
7.    x=1
8.  else
9.    x=2
10. endif
11. write(x)
```

```
1.  read(x)
2.  i = 1
3.  if (x==0)
4.    while (i<10)
5.      i=i+1
6.    endwhile
7.    x=1
8.  else
9.    x=2
10. endif
11. write(x)
```

- Slicing criterion <11,{x}>
- First program:
  - For x ==0, no termination (no output)
  - For x!=0, output 2
- Second program:
  - For x ==0, output 1
  - For x!=0, output 2

Such a slicer is impossible !!

So behavior equivalence is limited to only terminating executions of original program

Slicing computation and termination

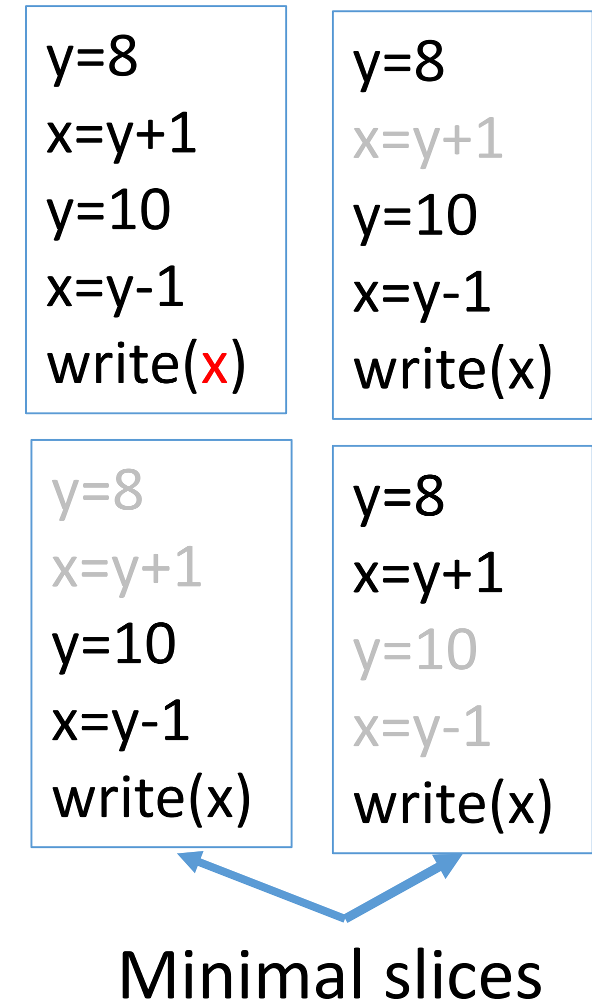
```
1.  read(x)
2.  i = 1
3.  if (x==0)
4.     while (i<10)
5.        i=i+0
6.     endwhile
7.     x=1
8.  else
9.     x=2
10. endif
11. write(x)
```

```
1.  read(x)
2.  i = 1
3.  if (x==0)
4.     while (i<10)
5.        i=i+1
6.     endwhile
7.     x=1
8.  else
9.     x=2
10. endif
11. write(x)
```

- Slicing criterion <11,{x}>
- First program:
  - For x ==0, no termination (no output)
  - For x!=0, output 2
  - Slice:
    - For x ==0, output 1
    - For x!=0, output 2
- Second program and slice:
  - For x ==0, output 1
  - For x!=0, output 2

Smallest possible slice ?

- Multiple slices as per the definition
  - Original program itself is always a slice
- Some may have more statements than other
- Smallest slice is one from which no more statements can be removed
  - *statement-minimum slice*
- Since multiple smallest slices of same size possible, these are statement-minimal slice
- Can it be computed?



Smallest possible slice computable ?

- Can a statement-minimal slice be computed?

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
    s =s + 1
    p = p * (i+1)
    x = x+2
endfor
// N>=0 → s=N
if (N>=0 && s != N)
   x= 1
endif
write(x)
```

Smallest possible slice computable ?

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
     s =s + 1
     p = p * (i+1)
     x = x+2
endfor
if (N>=0 && s != N)
    x= 1
endif
write(x)
```

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
     s =s + 1
     p = p * (i+1)
     x = x+2
endfor
if (N>=0 && s != N)
    x= 1
endif
write(x)
```

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
     s =s + 1
     p = p * (i+1)
     x = x+2
endfor
if (N>=0 && s != N)
    x= 1
endif
write(x)
```

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
     s =s + 1
     p = p * (i+1)
     x = x+2
endfor
if (N>=0 && s != N)
    x= 1
endif
write(x)
```

Smallest possible slice computable ?

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
     s =s + 1
     p = p * (i+1)
     x = x+2
endfor
if (N>=0 && s != N)
    x= 1
endif
write(x)
```

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
     s =s + 1
     p = p * (i+1)
     x = x+2
endfor
if (N>=0 && s != N)
    x= 1
endif
write(x)
```

Such a slicer is impossible !!

Smallest possible slice computable ?

```
read(N)
x=0
s=0
p=1
for (i=0; i < N; i++)
    s =s + 1
    p = p * (i+1)
    x = x+2
endfor
if (N>=0 && s != N)
   x= 1
endif
write(x)
```

"A practical definition of a minimal slice must avoid exact knowledge of the functions computed by pieces of code"

Such a slicer is impossible !!

Data flow information about a program is of this type

```
read(N)
x=0
s=N
p=1
for (i=0; i < N; i++)
    s =s + 2*i
    p = p * (i+1)
    x = x+2
endfor
if (N>0 && s != N*N)
   x= 1
endif
write(x)
```

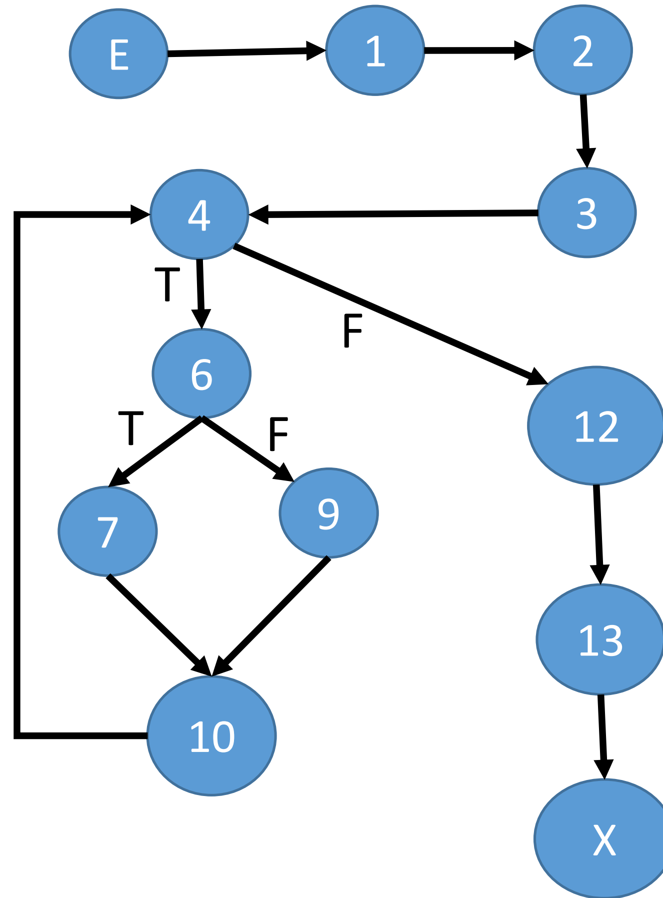
Some basic concepts

- Control flow graph
- Data dependence
- Control dependence

Control flow graph (CFG)

A graph in which nodes represent statements and the edges represent the control flow

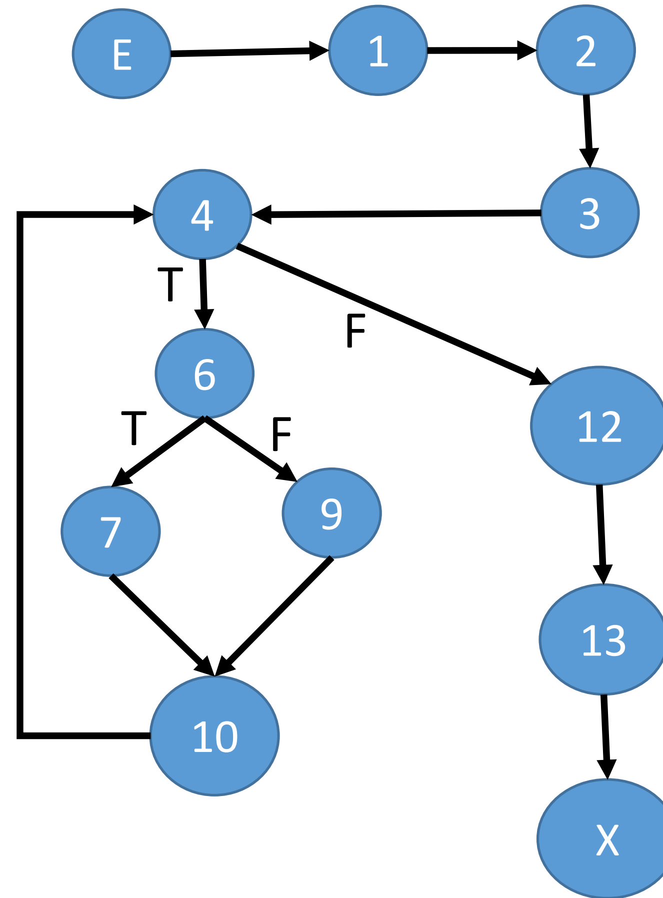
```
1.  s=0;
2.  p = 1;
3.  i = 0;
4.  while (i< 10)
5.  {
6.      if (a[i] < 0)
7.         p = p* a[i];
8.      else
9.         s = s + a[i];
10.     i++;
11. }
12. printf("%d", p);
13. printf("%d", s);
```



Data dependence

If S_2 potentially uses the value assigned by S_1 then S_2 is data dependent on S_1

```
1.  s=0;
2.  p = 1;
3.  i = 0;
4.  while (i< 10)
5.  {
6.      if (a[i] < 0)
7.          p = p* a[i];
8.      else
9.          s = s + a[i];
10.     i++;
11. }
12. printf("%d", p);
13. printf("%d", s);
```



Is node 4 data dependent on 2 ?

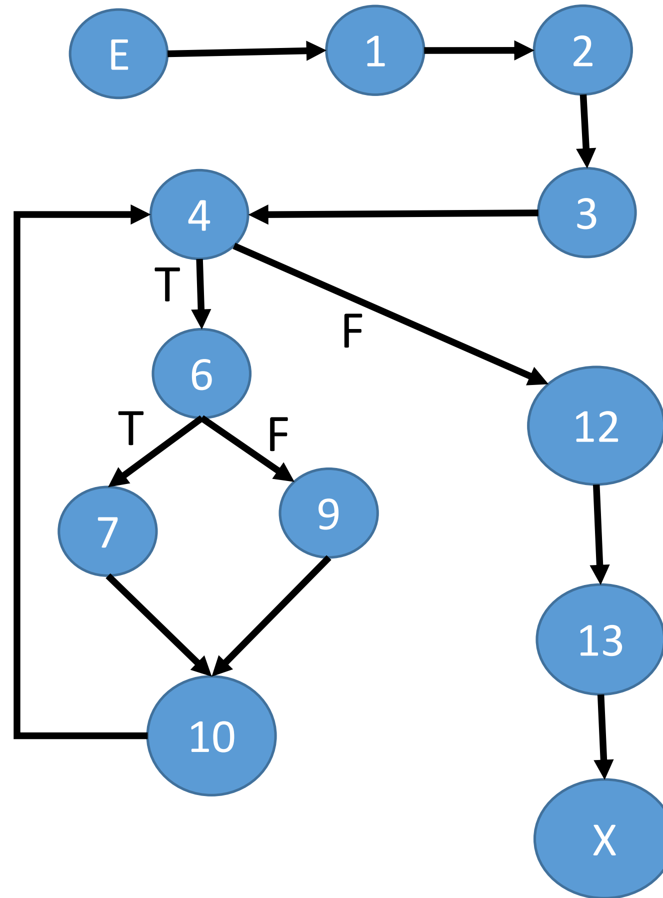
Is node 3 data dependent on 10 ?

Is node 6 data dependent on 10 ?

Data dependence

If S_2 potentially uses the value assigned by S_1 then S_2 is data dependent on S_1

```
1.  s=0;
2.  p = 1;
3.  i = 0;
4.  while (i< 10)
5.  {
6.      if (a[i] < 0)
7.         p = p* a[i];
8.      else
9.         s = s + a[i];
10.     i++;
11.}
12.printf("%d", p);
13.printf("%d", s);
```



Is node 4 data dependent on 2 ? NO

Is node 3 data dependent on 10 ? NO

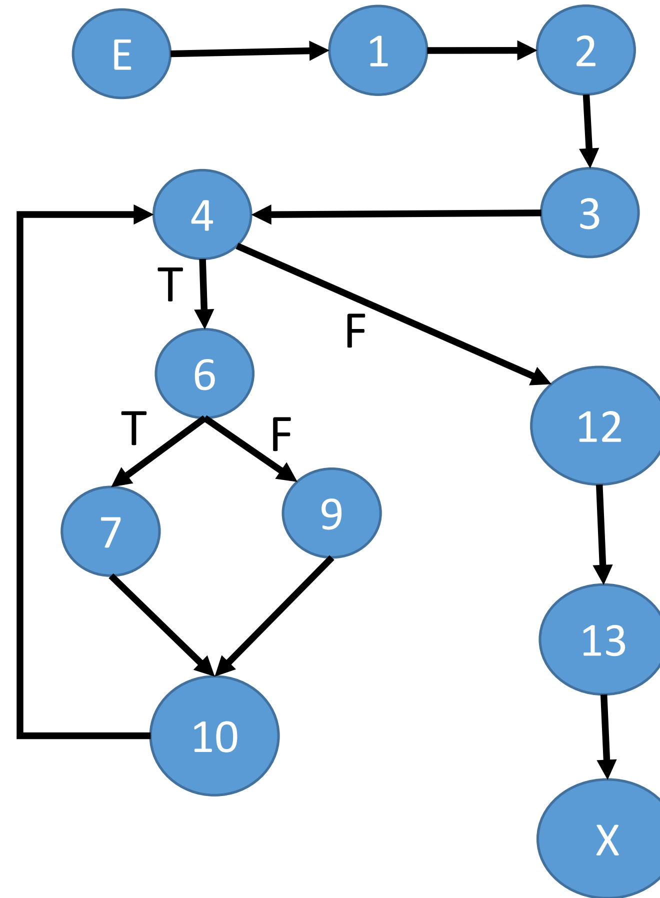
Is node 6 data dependent on 10 ? YES

If S_1 assigns to a variable x which is used by S_2 and there is a path from S_1 to S_2 along which there is no further assignment to x then S_2 is data dependent on S_1

Data dependence

If S_2 potentially uses the value assigned by S_1 then S_2 is data dependent on S_1

```
1.  s=0;
2.  p = 1;
3.  i = 0;
4.  while (i< 10)
5.  {
6.      if (a[i] < 0)
7.        p = p* a[i];
8.      else
9.        s = s + a[i];
10.     i++;
11.}
12.printf(“%d”, p);
13.printf(“%d”, s);
```



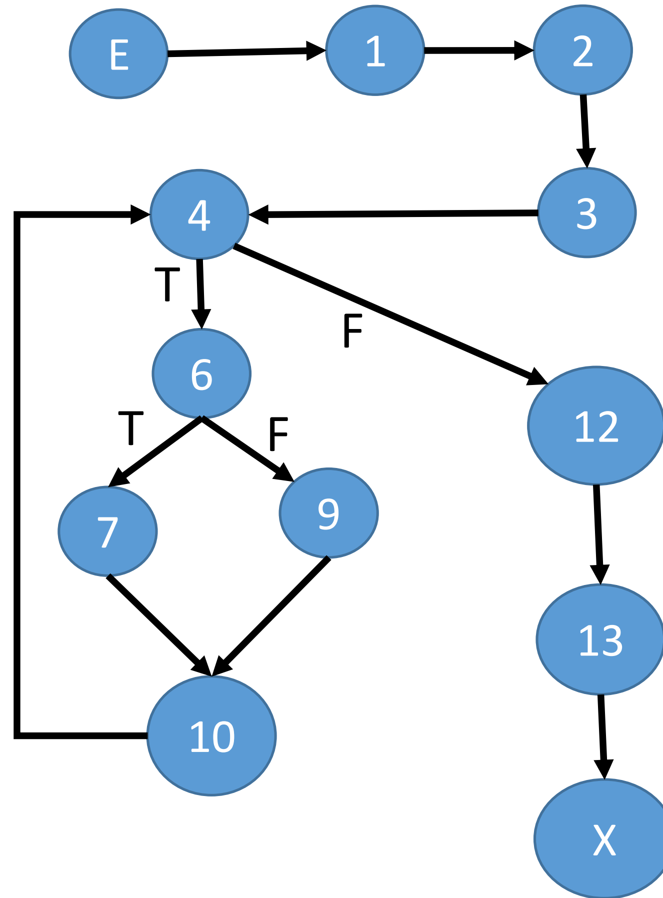
node 4 is data dependent on 3 and 10

node 7 is data dependent on 2, 7, 3 and 10

Control dependence

If S_1 decides whether S_2 will be executed or not then S_2 is control dependent on S_1

```
1.  s=0;
2.  p = 1;
3.  i = 0;
4.  while (i< 10)
5.  {
6.      if (a[i] < 0)
7.          p = p* a[i];
8.      else
9.          s = s + a[i];
10.     i++;
11.}
12.printf("%d", p);
13.printf("%d", s);
```

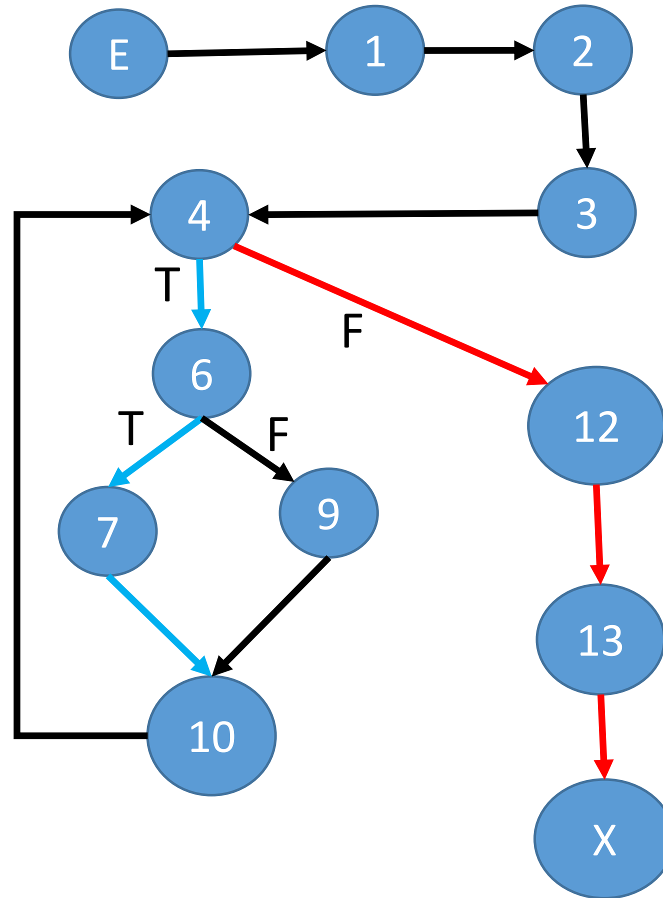


Is node 10 control dependent on 4 ?

Control dependence

If S_1 decides whether S_2 will be executed or not then S_2 is control dependent on S_1

```
1.  s=0;
2.  p = 1;
3.  i = 0;
4.  while (i< 10)
5.  {
6.      if (a[i] < 0)
7.         p = p* a[i];
8.      else
9.         s = s + a[i];
10.     i++;
11.}
12.printf("%d", p);
13.printf("%d", s);
```

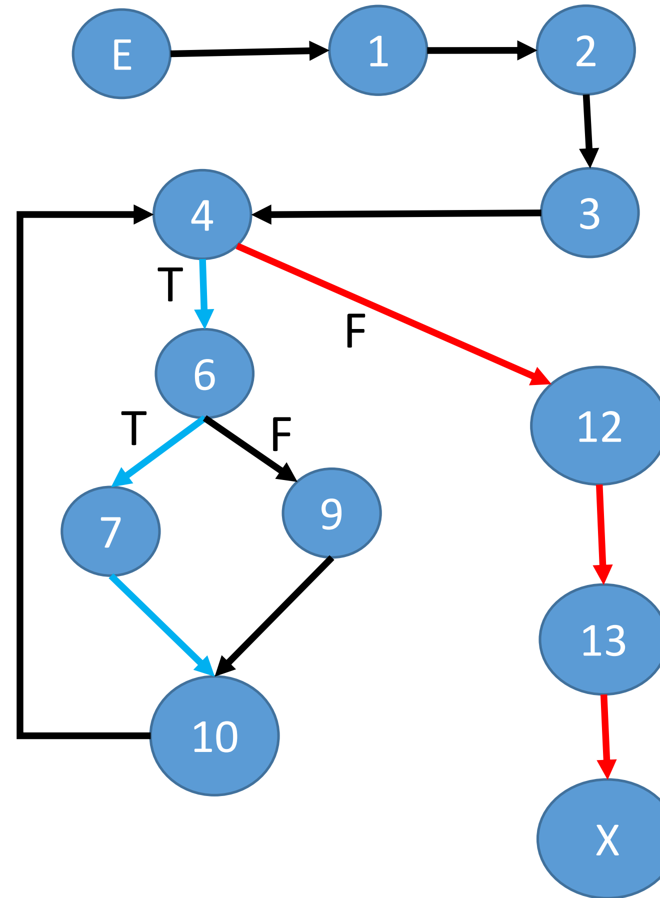


Is node 10 control dependent on 4 ? Yes

Basic concepts : Control dependence

If S_1 decides whether S_2 will be executed or not then S_2 is control dependent on S_1

```
1.  s=0;
2.  p = 1;
3.  i = 0;
4.  while (i< 10)
5.  {
6.      if (a[i] < 0)
7.         p = p* a[i];
8.      else
9.         s = s + a[i];
10.     i++;
11.}
12.printf("%d", p);
13.printf("%d", s);
```



Is node 10 control dependent on 4 ? Yes

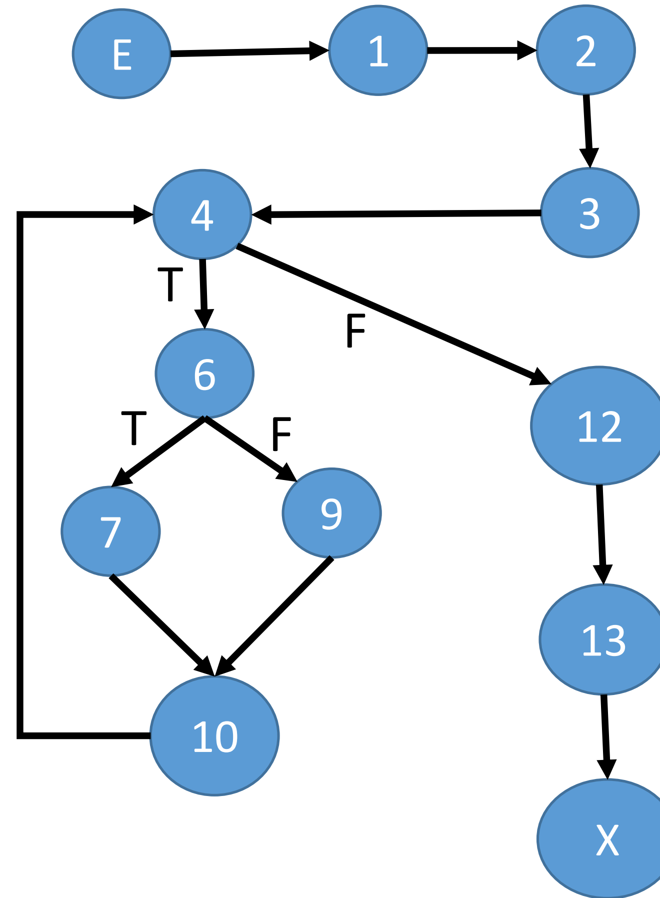
Outgoing edges from S_1 can be partitioned in two non-empty sets, say E_1 and E_2 such that :

- All paths to EXIT from E_1 avoid S_2
- All paths to EXIT from E_2 go through S_2

Basic concepts : Control dependence

If S_1 decides whether S_2 will be executed or not then S_2 is control dependent on S_1

```
1.  s=0;
2.  p = 1;
3.  i = 0;
4.  while (i< 10)
5.  {
6.      if (a[i] < 0)
7.          p = p* a[i];
8.      else
9.          s = s + a[i];
10.     i++;
11. }
12. printf("%d", p);
13. printf("%d", s);
```



Nodes 6 and 10 are control dependent on 4
Nodes 7 and 9 are control dependent on 6

As a special case, nodes 1,2,3,4,12,13 are taken to be control dependent on ENTRY

Slice computation from data flow : example

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

Slice computation from data flow : example

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

Slice computation from data flow : example

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

Slice computation from data flow : example

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

Slice computation from data flow : example

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

```
read(n)
i = 1
s = 0
p = 1
j = 1
while i<=n do
   s = s + j
   p = p * j
   i = i + 1
   j = j + 1
endwhile
write(s)
write(p)
```

Program slicing as data flow equations

C : <s, V> : Slicing criterion, **B(S)** : Branch conditions which control statements in S
DEF(n) : Variables which may be changed at n , **REF(n)** : Variables which may be referred at n
BC(b) : Criterion <b, REF(b)> for branching statement b , **R** : Relevant var set, **S** : Slice

$$R_{in}[C](n) = (R_{out}[C](n) - DEF(n)) \cup (if\ DEF(n) \cap R_{out}[C](n) \neq \phi\ then\ REF(n)\ else\ \phi) \cup (if\ n = s\ then\ V\ else\ \phi)$$

$$R_{out}[C](n) = \bigcup_{for\ all\ successors\ m\ of\ n} R_{in}[C](m)$$

$$R^0[C](n) = R_{out}[C](n)$$

$$S^0[C] = \{\, n \mid R^0[C](n) \cap DEF(n) \neq \phi \}$$

$$R^1[C](n) = R^0[C](n) \bigcup_{b \in B(S^0[C] \cup \{s\})} R^0[BC(b)](n)$$

$$R^{k+1}[C](n) = R^k[C](n) \bigcup_{b \in B(S^k[C])} R^0[BC(b)](n)$$

$$S^{k+1}[C] = \{\, n \mid R^{k+1}(n) \cap DEF(n) \neq \phi\ OR\ n \in B(S^k[C]) \}$$

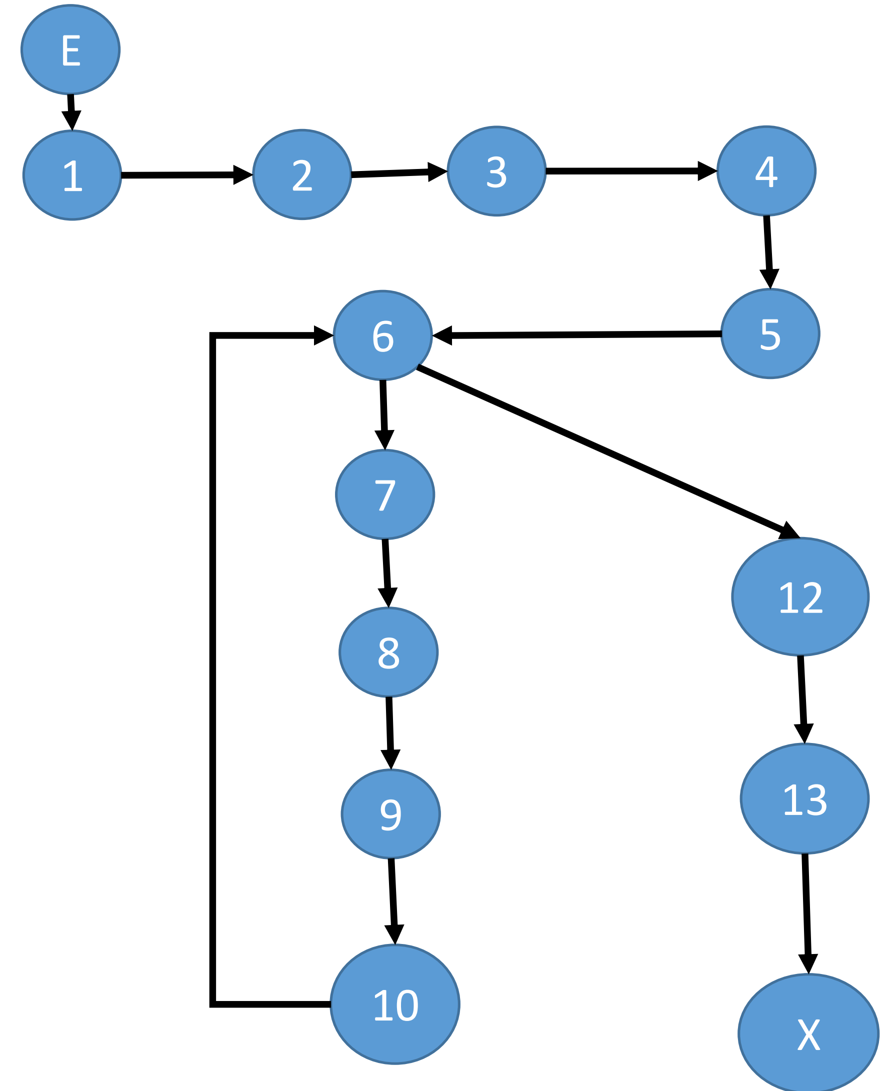
$$S[C] = \bigcup_{k \geq 0} S^k[C]$$

Example

```
1.   read(n)
2.   i = 1
3.   s = 0
4.   p = 1
5.   j = 1
6.   while i<=n do
7.      s = s + j
8.      p = p * j
9.      i = i + 1
10.     j = j + 1
11. endwhile
12. write(s)
13. write(p)
```

N	DEF	REF	$R^0[C]$	$R^0[b]$	$R^1[C]$
1	n			n	n
2	i			i,n	i,n
3	s			i,n	i,n
4	p		p	i,n	p,i,n
5	j		p,j	i,n	p,j,i,n
6		i,n	p,j	i,n	p,j,i,n
7	s	s,j	p,j	i,n	p,j,i,n
8	p	p,j	p,j	i,n	p,j,i,n
9	i	i	p,j	i,n	p,j,i,n
10	j	j	p,j	i,n	p,j,i,n
12			p		p

$S^0[C] = \{4,5,8,10\}, B(S^0[C]) = \{6\}, S^1[C] = \{1,2,4,5,6,8,9,10\}$

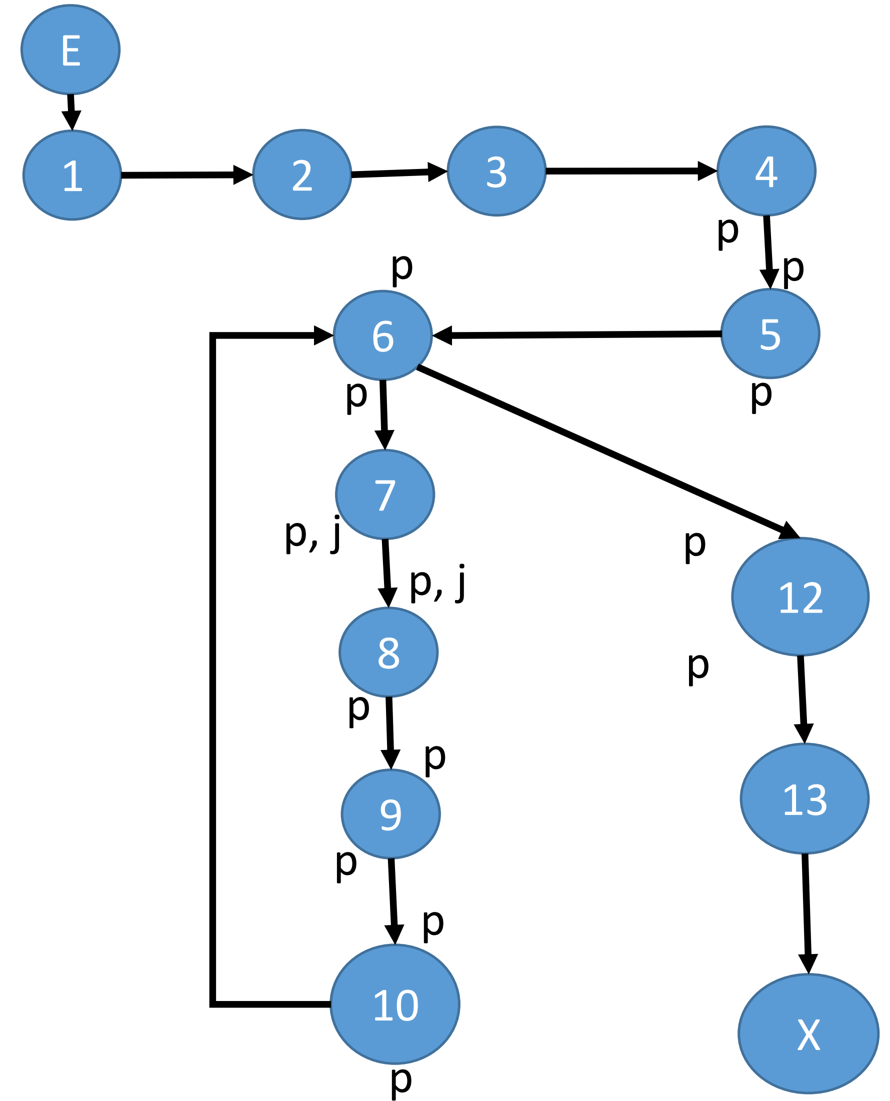


Example

```
1.  read(n)
2.  i = 1
3.  s = 0
4.  p = 1
5.  j = 1
6.  while i<=n do
7.      s = s + j
8.      p = p * j
9.      i = i + 1
10.     j = j + 1
11. endwhile
12. write(s)
13. write(p)
```

N	DEF	REF	$R^0[C]$	$R^0[b]$	$R^1[C]$
1	n			n	n
2	i			i,n	i,n
3	s			i,n	i,n
4	p		p	i,n	p,i,n
5	j		p,j	i,n	p,j,i,n
6		i,n	p,j	i,n	p,j,i,n
7	s	s,j	p,j	i,n	p,j,i,n
8	p	p,j	p,j	i,n	p,j,i,n
9	i	i	p,j	i,n	p,j,i,n
10	j	j	p,j	i,n	p,j,i,n
12			p		p

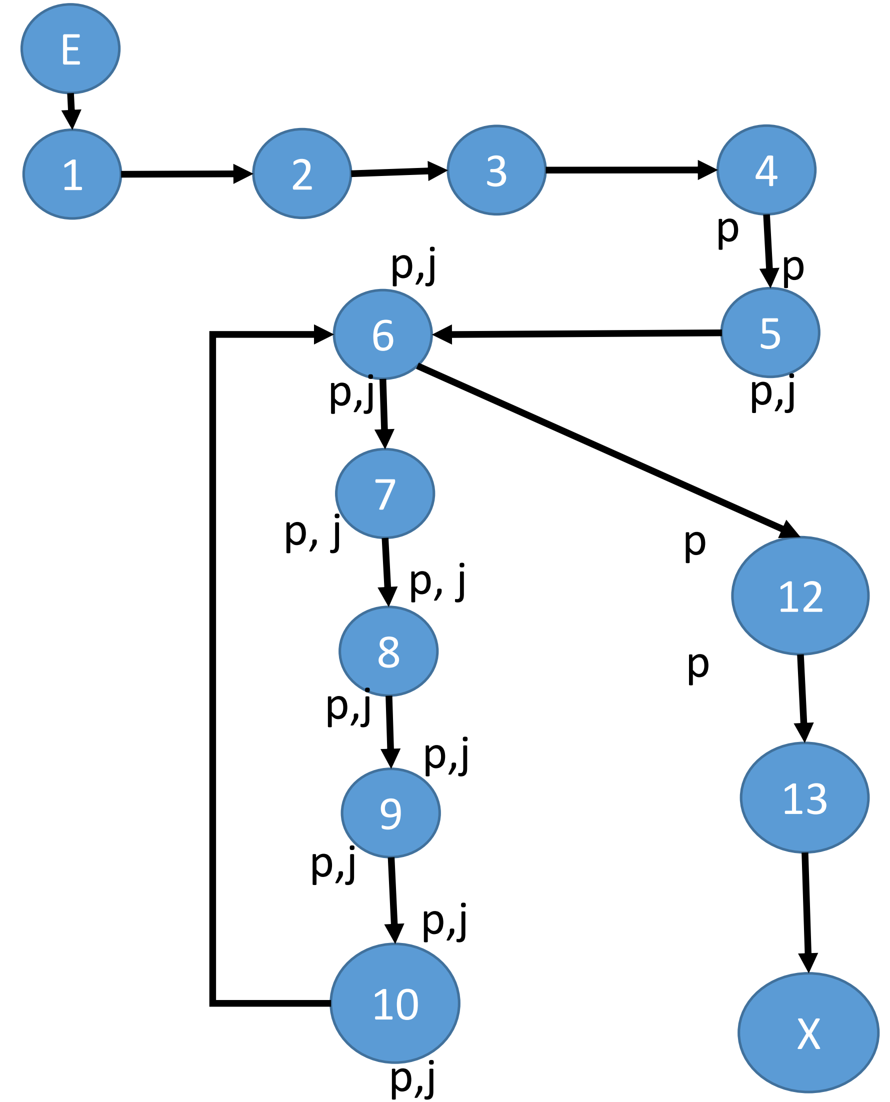
$S^0[C] = \{4,5,8,10\}, B(S^0[C]) = \{6\}, S^1[C] = \{1,2,4,5,6,8,9,10\}$



Example

```
1.  read(n)
2.  i = 1
3.  s = 0
4.  p = 1
5.  j = 1
6.  while i<=n do
7.     s = s + j
8.     p = p * j
9.     i = i + 1
10.    j = j + 1
11. endwhile
12. write(s)
13. write(p)
```

N	DEF	REF	$R^0[C]$	$R^0[b]$	$R^1[C]$
1	n			n	n
2	i			i,n	i,n
3	s			i,n	i,n
4	p		p	i,n	p,i,n
5	j		p,j	i,n	p,j,i,n
6		i,n	p,j	i,n	p,j,i,n
7	s	s,j	p,j	i,n	p,j,i,n
8	p	p,j	p,j	i,n	p,j,i,n
9	i	i	p,j	i,n	p,j,i,n
10	j	j	p,j	i,n	p,j,i,n
12			p		p

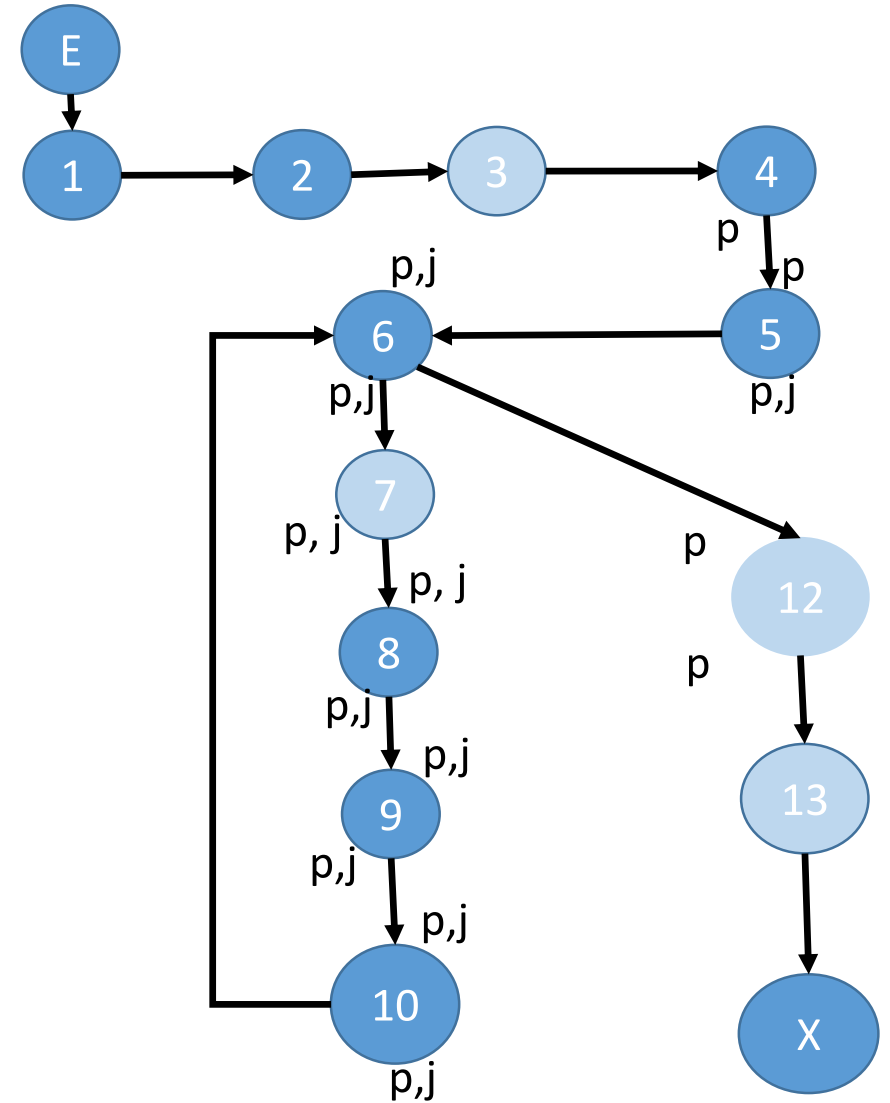


$S^0[C] = \{4,5,8,10\}, B(S^0[C]) = \{6\}, S^1[C] = \{1,2,4,5,6,8,9,10\}$

Example

1. read(n)
2. i = 1
3. s = 0
4. p = 1
5. j = 1
6. while i<=n do
7. s = s + j
8. p = p * j
9. i = i + 1
10. j = j + 1
11. endwhile
12. write(s)
13. write(p)

N	DEF	REF	$R^0[C]$	$R^0[b]$	$R^1[C]$
1	n			n	n
2	i			i,n	i,n
3	s			i,n	i,n
4	p		p	i,n	p,i,n
5	j		p,j	i,n	p,j,i,n
6		i,n	p,j	i,n	p,j,i,n
7	s	s,j	p,j	i,n	p,j,i,n
8	p	p,j	p,j	i,n	p,j,i,n
9	i	i	p,j	i,n	p,j,i,n
10	j	j	p,j	i,n	p,j,i,n
12			p		p

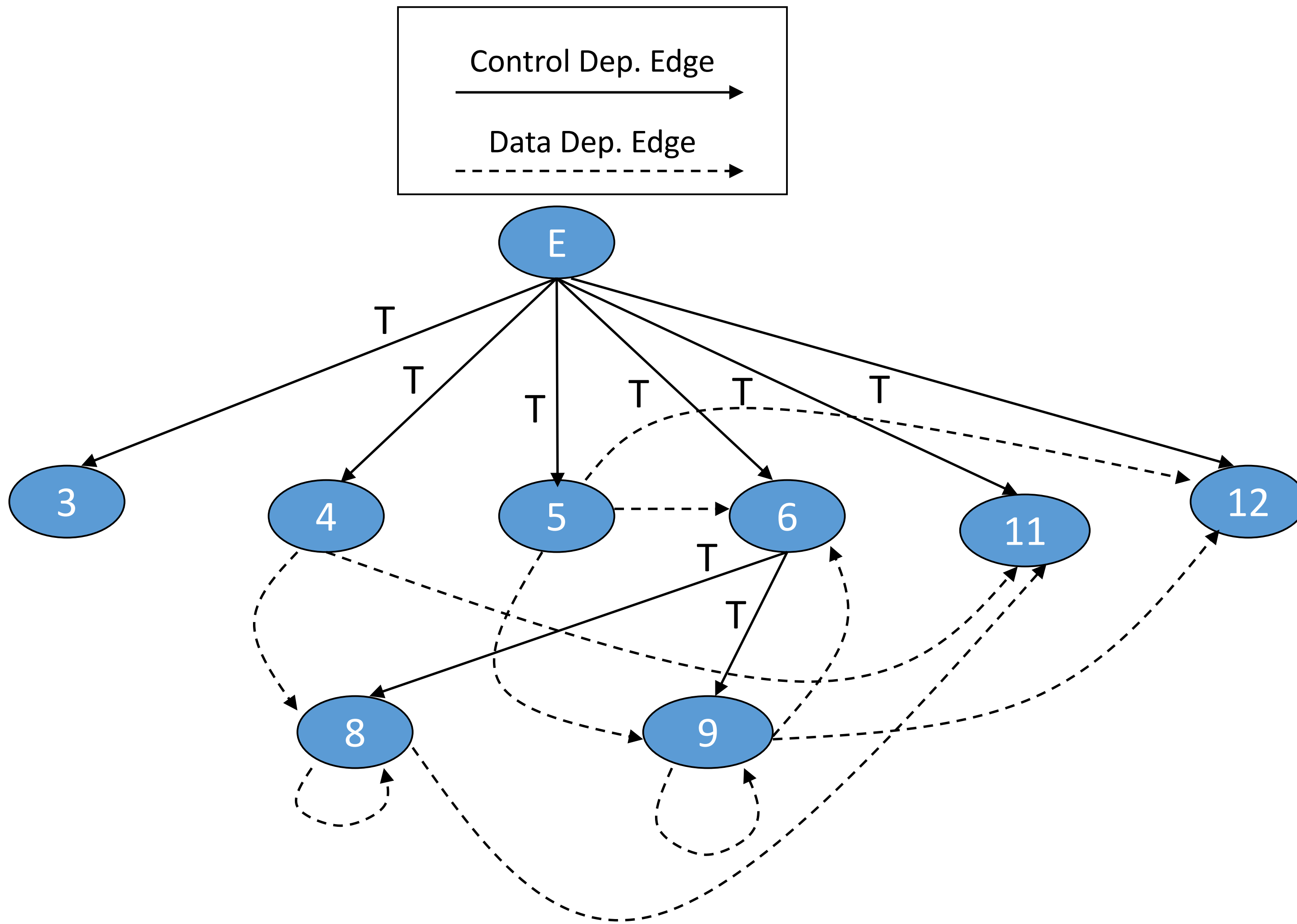


$S^0[C] = \{4,5,8,10\}, B(S^0[C]) = \{6\}, S^1[C] = \{1,2,4,5,6,8,9,10\}$

Program slicing using PDG

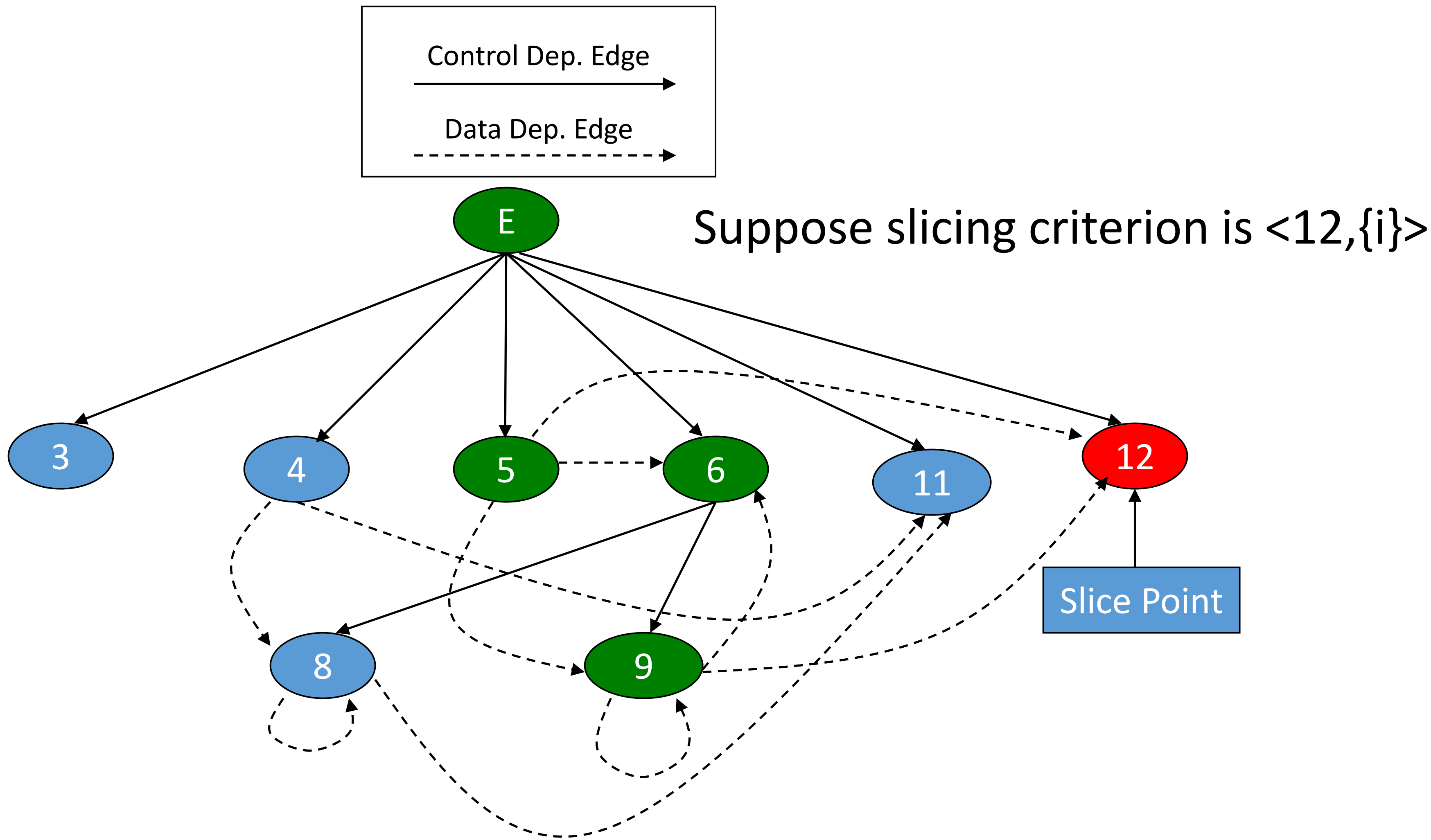
- Program dependence graph (PDG)
  - A multi graph
  - Combines the data dependence and control dependence
  - Nodes are same as in CFG
  - Edges indicate data or control dependence
- Proposed by Ottenstien & Ottenstein (1984)
  - Casted program slicing as a graph reachability problem over PDG

Program slicing using PDG




```
1.  main()
2.  {
3.    int i, sum;
4.    sum = 0
5.    i= 1
6.    while (i<=10)
7.    {
8.      sum = sum + 1
9.      i = i + 1
10.   }
11.   printf("%d", sum)
12.   printf("%d\n", i)
13. }
```

Program slicing using PDG




```
1.   main()
2.   {
3.     int i, sum;
4.     sum = 0
5.     i= 1
6.     while (i<=10)
7.     {
8.        sum = sum + 1
9.        i = i + 1
10.   }
11.   printf(“%d”, sum)
12.   printf(“%d\n”, i)
13. }
```

Nodes from which slice point node is reachable, belong to slice

Use of backward slice

- Debugging
  - Understanding the cause of the bug
- Program understanding
  - To focus on understanding only one functionality at a time
- Eliminating irrelevant piece of code
  - To have smaller size code for model checking

To summarise …

- Executable backward static slicing by Weiser (as program slicing)
    - Slice P’ = S(P, <s, X>) is result of deleting some statements from P, such that

      P and P’ are indistinguishable when their behaviour is observed through “same corresponding window”, <s,X> and <s’,X> respectively

        For all corresponding terminating executions of P and P’ (on same input)
        Value of X at s in execution of P is same as value of X at s’ in execution of P’

- Computed as a data flow equation or by backward traversal over PDG
    - PDG is a graph representing data dependence as well as control dependence relationship among statements
- A minimal slice is non-computable