Course Notes Set 9:
Review of Graphs

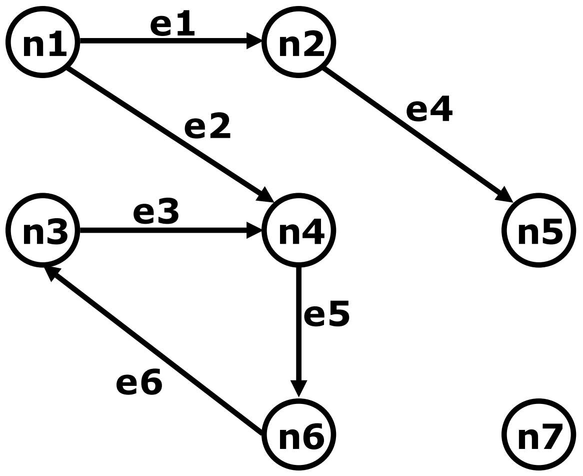
Computer Science and Software Engineering
Auburn University

Graphs

- Graph theory is a branch of topology, so is a mathematically rigorous subject.
- For our purposes, however, we will only need to understand graphs at the level presented in an undergraduate data structures and algorithms course.
- Although undirected graphs are the more general concept, we will only discuss directed graphs.

Directed Graphs

- A directed graph (or digraph) G = (V, E) consists of a finite set V = {n_1, n_2, …, n_m} of nodes and a finite set E = {e_1, e_2, …, e_p} of edges, where each edge e_k = {n_i, n_j} is an ordered pair (start-node, terminal-node) of nodes from V.

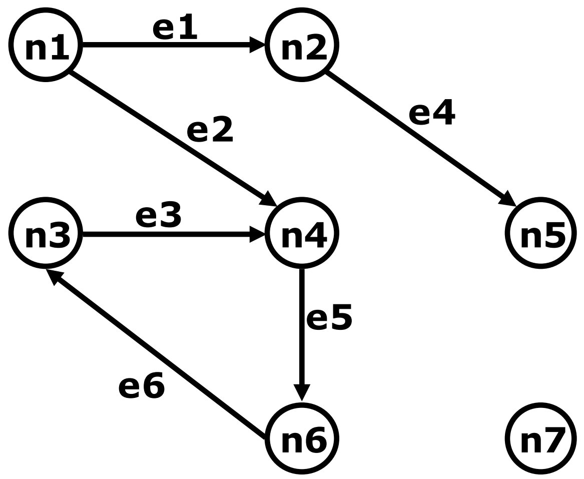


V = {n1, n2, n3, n4, n5, n6, n7}

E = {e1, e2, e3, e4, e5, e6}
= {(n1, n2), (n1, n4), (n3, n4), (n2, n5), (n4, n6), (n6, n3)}

Nodes and Degrees

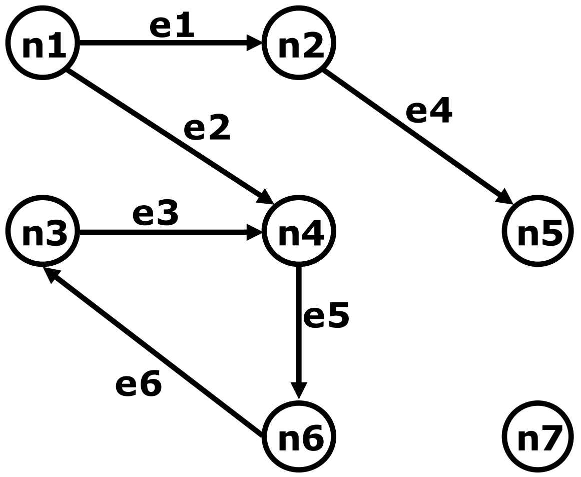
- **Indegree(node)** = number of distinct edges that have the node as a terminal node.
- **Outdegree(node)** = number of distinct edges that have the node as a start node.
- **Source node** = a node with indegree 0.
- **Sink node** = a node with outdegree 0.
- **Transfer node** = a node with indegree != 0 and outdegree != 0.



Indegree(n1) = 0
Indegree(n2) = 1
Indegree(n4) = 2
Indegree(n7) = 0
Outdegree(n1) = 2
Outdegree(n2) = 1
Outdegree(n4) = 1
Outdegree(n7) = 0

Paths and Semi-Paths

- **Directed Path** = a sequence of edges such that, for any adjacent pair of edges e_i, e_j in the sequence, the terminal node of the first edge is the start node of the second edge.
- **Directed Semi-path** = a sequence of edges such that, for at least one adjacent pair of edges ei, ej in the sequence, the initial node of the first edge is the initial node of the second edge, or the terminal node of the first edge is the terminal node of the second edge.



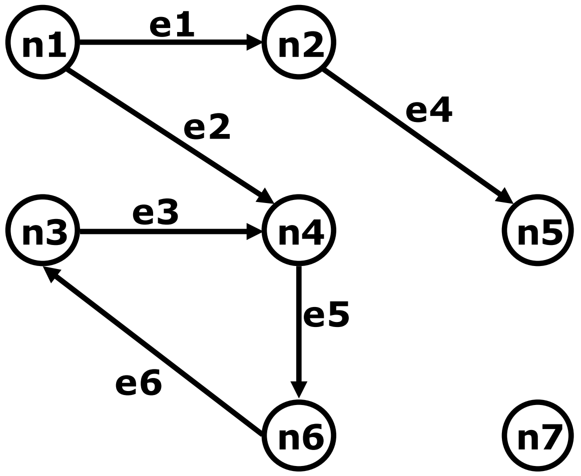
There is a path from n1 to n6.
There is a semi-path between n1 and n3.
There is a semi-path between n2 and n4.
There is a semi-path between n5 and n6.

Connectedness

Two nodes n_i and n_j in a directed graph are:

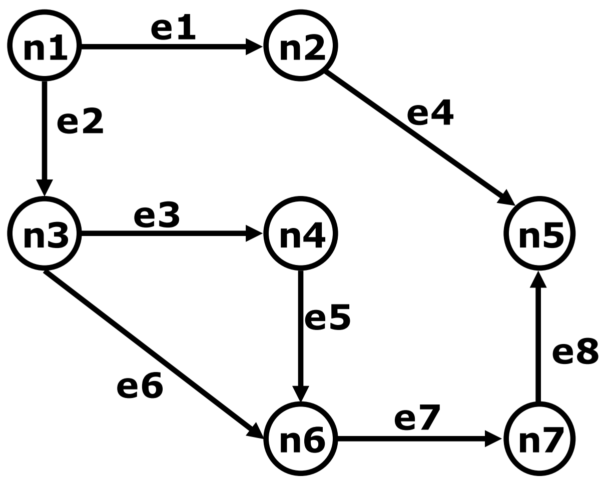
- **0-connected** iff there is no path or semi-path between n_i and n_j.
- **1-connected** iff there is a semi-path but no path between n_i and n_j.
- **2-connected** iff there is a path between n_i and n_j.
- **3-connected** iff there is a path from n_i to n_j and a path from n_j to n_i.

A graph is **connected** iff all pairs of nodes are either 1-connected or 2-connected. A graph is **strongly connected** iff all pairs of nodes are 3-connected.

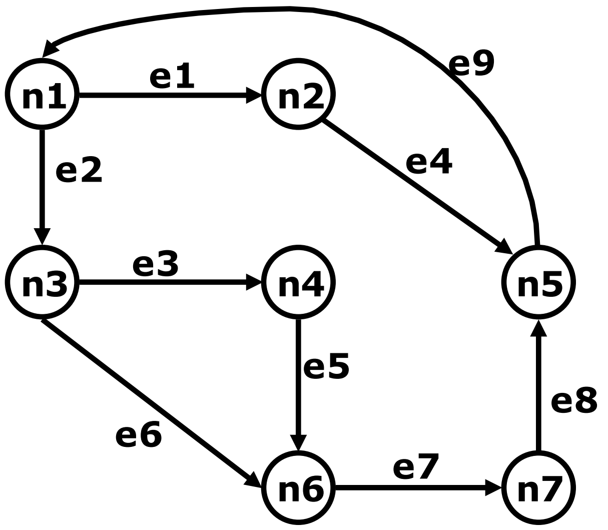


n1 and n7 are 0-connected.
n2 and n6 are 1-connected.
n1 and n6 are 2-connected.
n3 and n6 are 3-connected.
The graph is not connected.
The graph is not strongly connected.

Connectedness



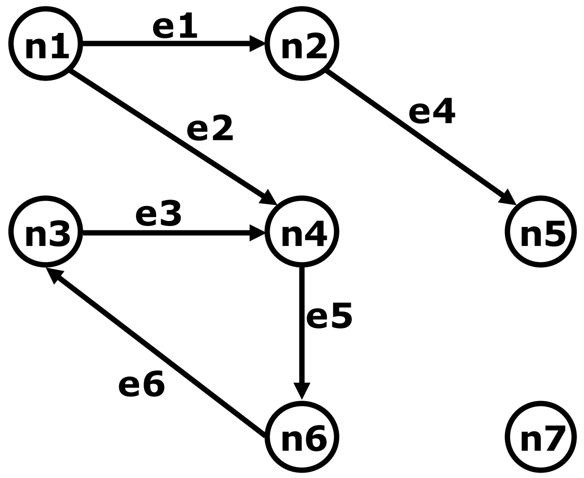
Connected, but not strongly connected.



Strongly connected.

Strong Components

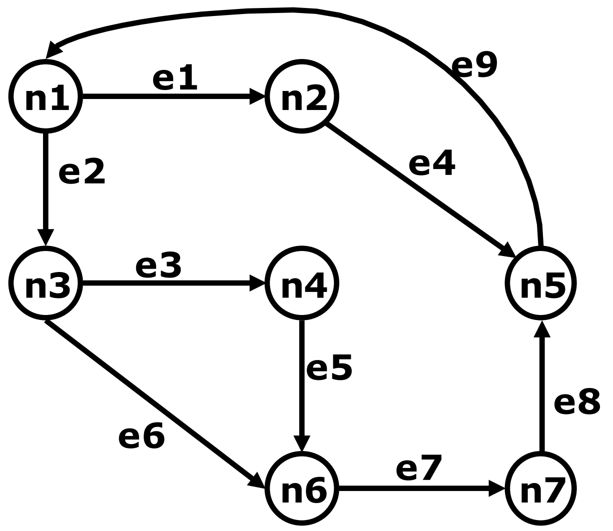
- A strong component of a directed graph is a maximal set of 3-connected nodes; that is, a maximal set of strongly connected nodes.



{n3, n4, n6} is a strong component.
{n7} is a strong component.

Cyclomatic Number

- The cyclomatic number of a *strongly connected directed graph* G = (V, E) is given by v(G) = |E| - |V| + p, where p is the number of strong components in the graph.
- v(G) is also equal to the number of bounded areas defined by the graph.



v(G) = 9 – 7 + 1 = 3

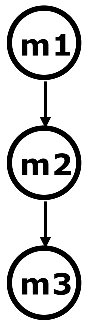
Program Graphs

- Given a program written in an imperative programming language, its program graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represent flow of control.
- At the module level, program graphs should be connected. Loops will define strongly connected components.

Graphs for Control Constructs

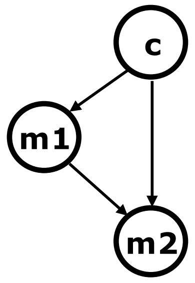
Sequence

```
{
    m1();
    m2();
    m3();
}
```



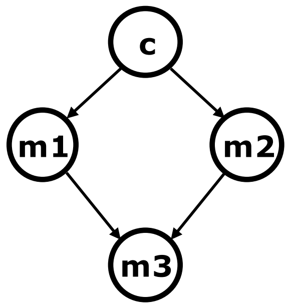
if-then

```
if (c) {
    m1();
}
m2();
```

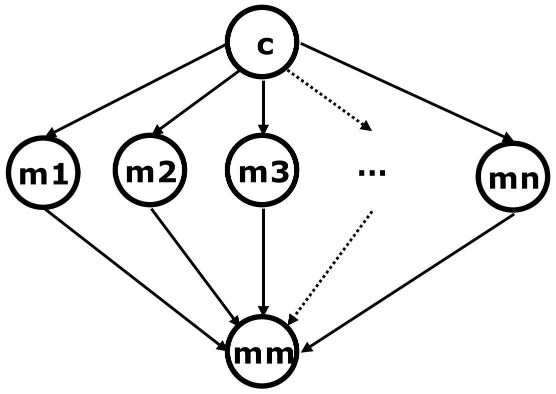


if-then-else

```
if (c) {
    m1();
}
else {
    m2();
}
m3();
```



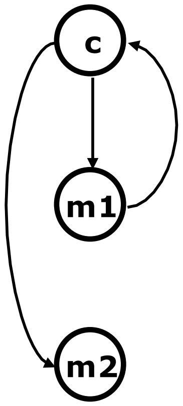
Switch/multi-way decision



Graphs for Control Constructs

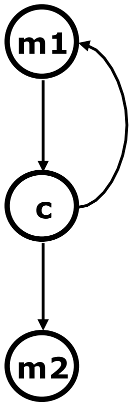
Pre-test Loop

```
while (c) {
   m1();
}
m2();
```



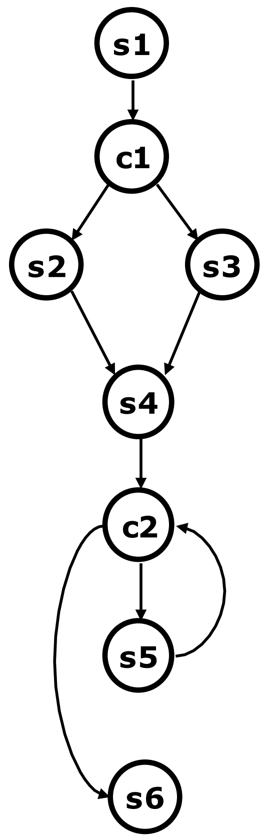
Post-test Loop

```
do {
   m1();
} while (c);
m2();
```



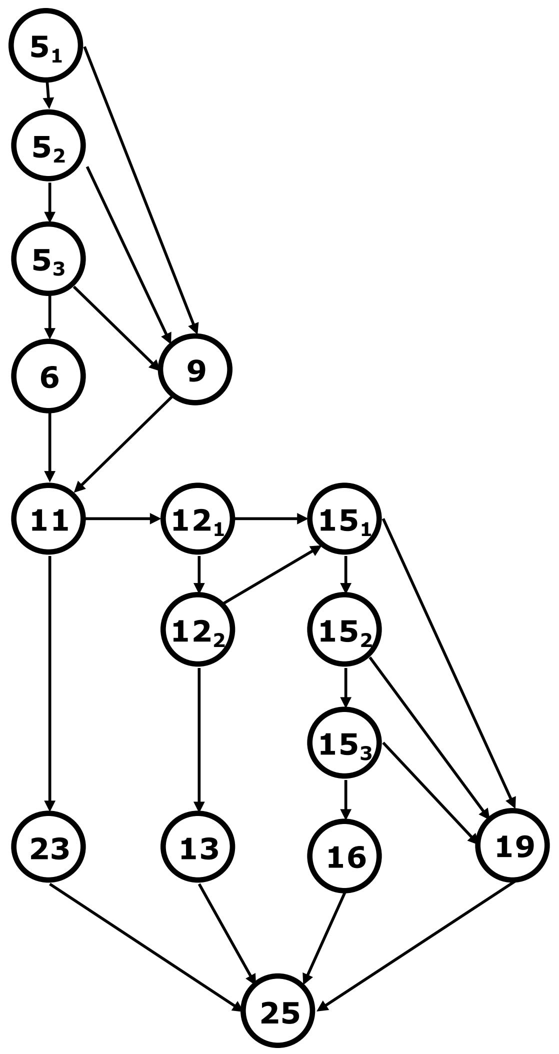
Program Graph Example

```
s1();
if (c1) {
   s2();
}
else {
   s3();
}
s4();
while (c2) {
   s5();
}
s6();
```



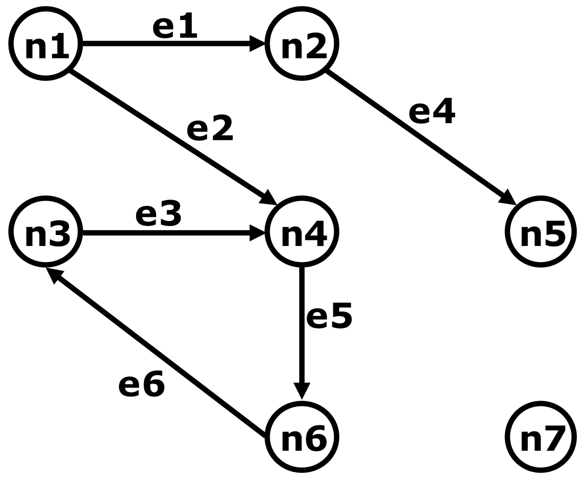
Program Graph Example

```
 2  public static String triangle (int a, int b, int c) {
 3      String result;
 4      boolean isATriangle;
 5      if ( ( a < b + c ) && ( b < a + c ) && ( c < a + b ) ) {
 6          isATriangle = true;
 7      }
 8      else {
 9          isATriangle = false;
10      }
11      if ( isATriangle ) {
12          if ( ( a == b ) && ( b == c ) ) {
13              result = "Triangle is equilateral.";
14          }
15          else if ( ( a != b ) && ( a != c ) && ( b != c ) ) {
16              result = "Triangle is scalene.";
17          }
18          else {
19              result = "Triangle is isosceles.";
20          }
21      }
22      else {
23          result = "Not a triangle.";
24      }
25      return result;
26  }
```

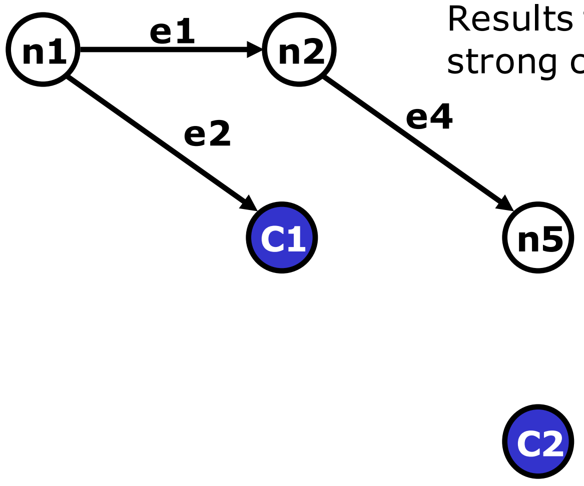


Condensation Graph

- Given a graph G = (V, E), its condensation graph is formed by replacing some components with a condensing node.

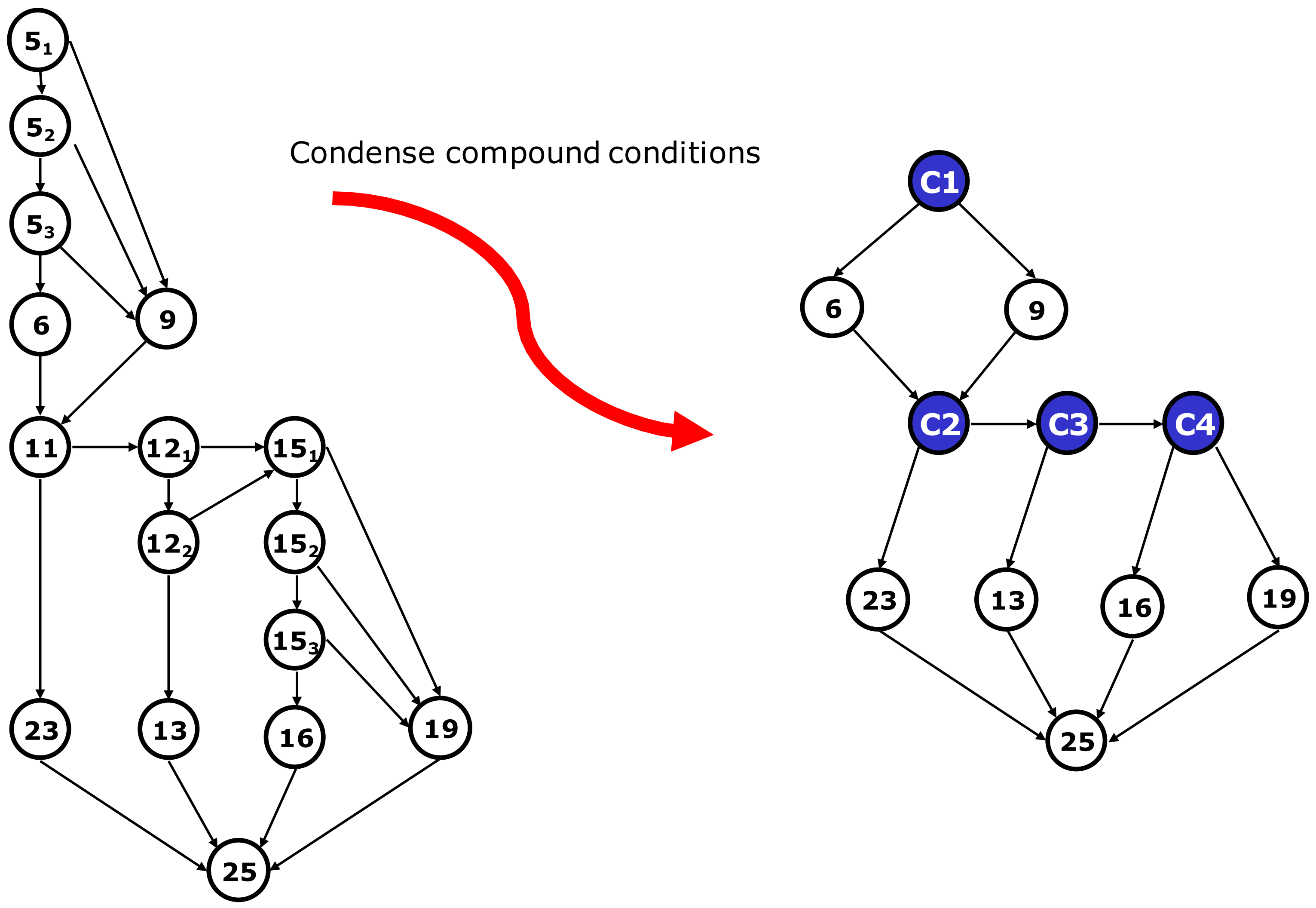


Condensation Graph:



Results from condensing strong components.

Condensation Graph


Condense compound conditions
5_1
5_2
5_3
6
9
11
12_1
12_2
15_1
15_2
15_3
23
13
16
19
25
C1
6
9
C2
C3
C4
23
13
16
19
25