Graphs
(chapter 14)

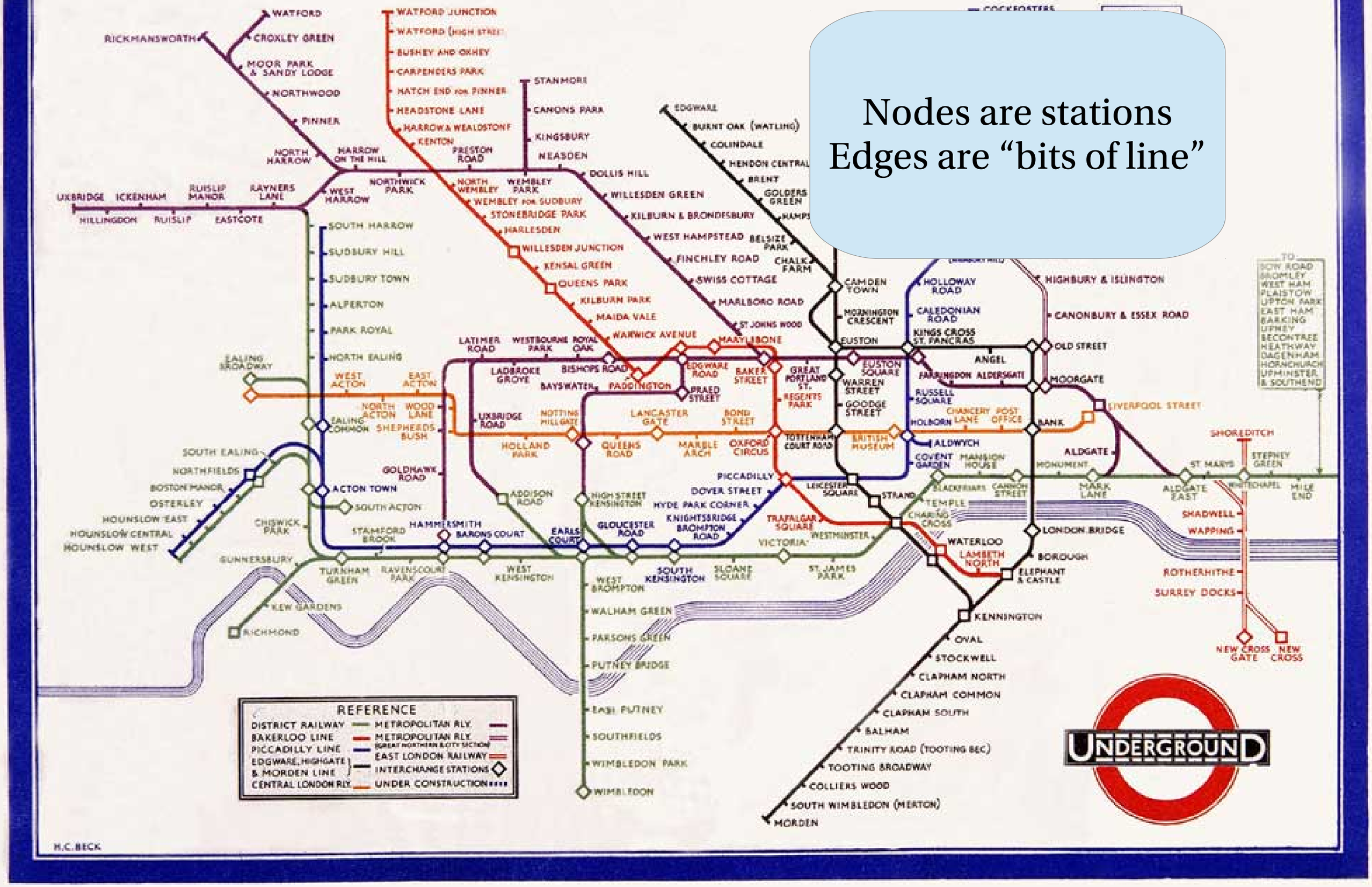
Terminologi

En graf är en datastruktur som består av en mängd *noder* (vertices) och en mängd *bågar* (edges)

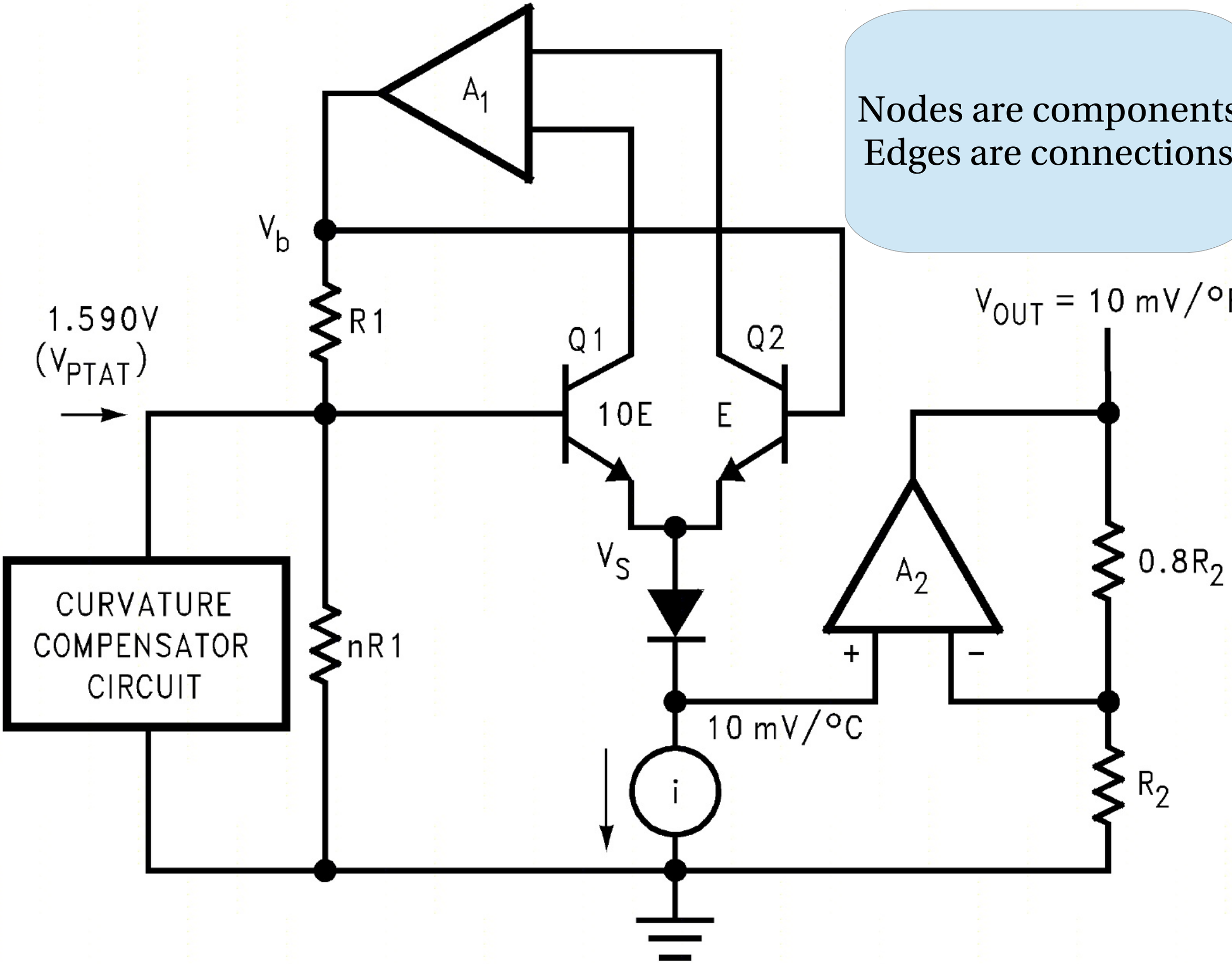
- en båge är ett par (*a*, *b*) av två noder
- en båge kan vara cyklisk — peka på sig själv

- en graf kan vara *riktad* eller *oriktad*
  - om den är oriktad så är (*a*, *b*) och (*b*, *a*) samma sak
- en graf kan vara *viktad* eller *oviktad*
  - om den är viktad så är bågarna tripler (*a*, *b*, *w*), där *w* är vikten (oftast ett tal > 0)

Nodes are stations
Edges are “bits of line”

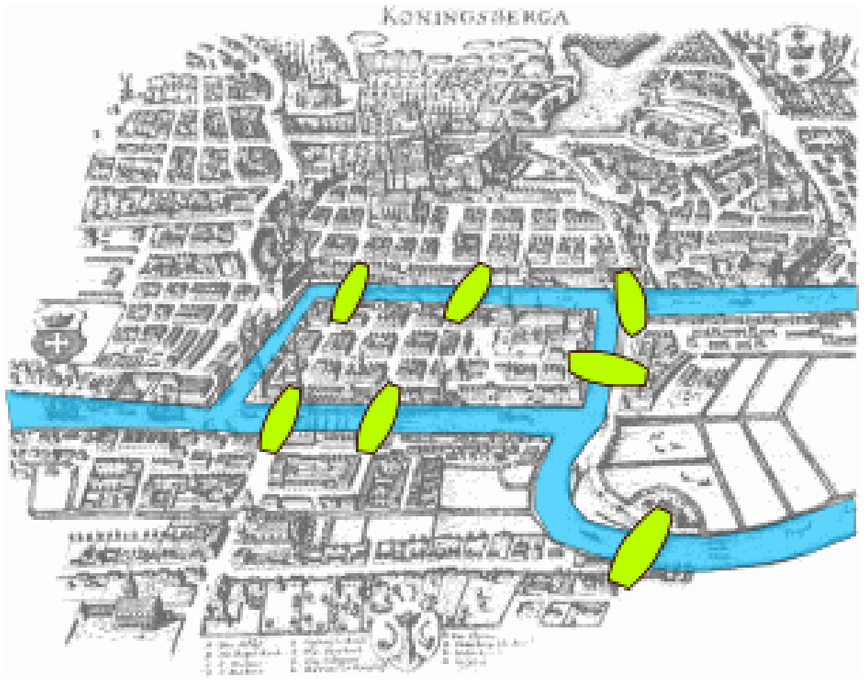


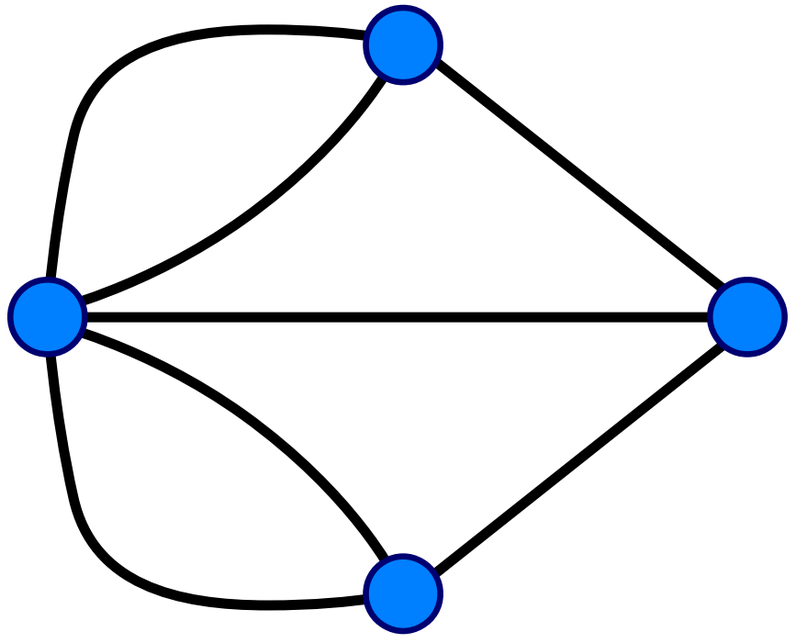
Nodes are components
Edges are connections

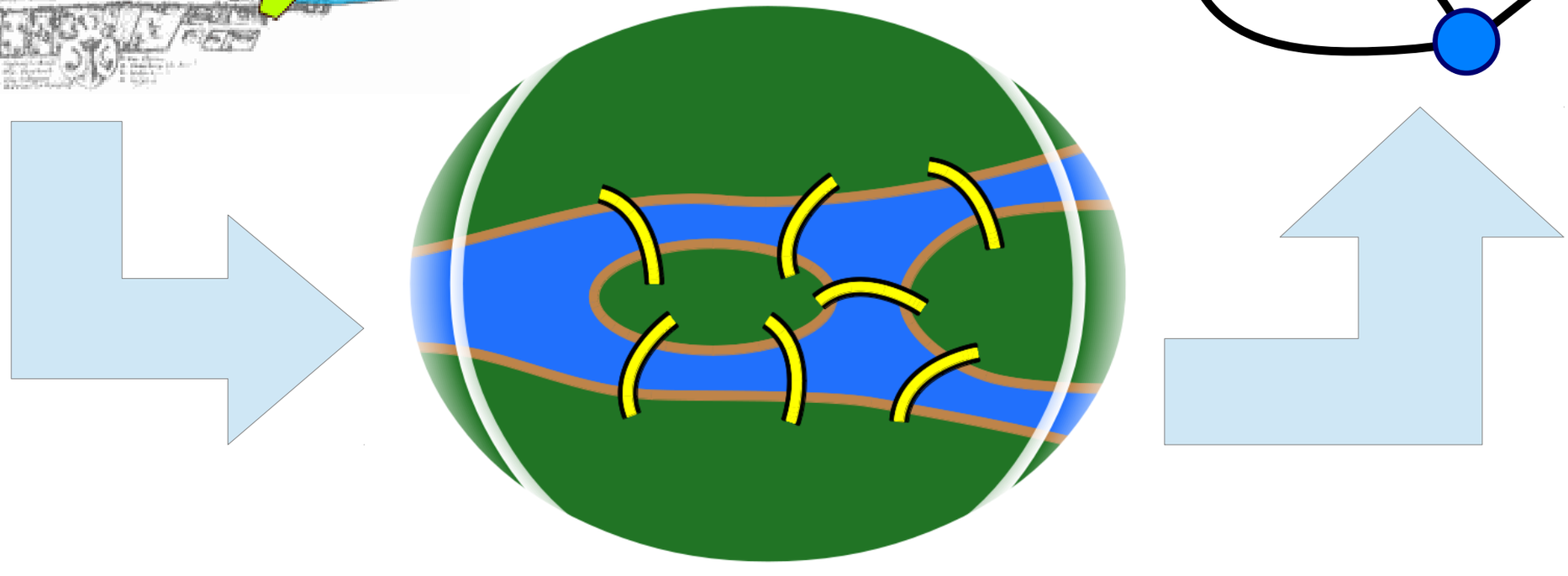


Seven bridges of Königsberg

http://en.wikipedia.org/wiki/Seven_Bridges_of_Königsberg







Grafer

Träd är begränsade — de kan ha endast en förälder

- grafer kan ha hur många som helst

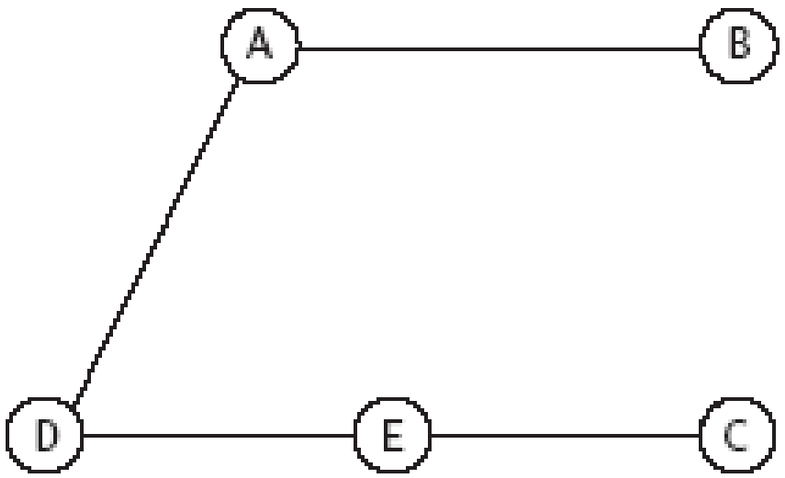
Grafer och grafalgoritmer används för:

- stora kommunikationsnätverk
- algoritmer som får Internet att funka
- att beräkna den optimala placeringen av komponenter på integrerade kretsar
- att beskriva vägnät, kartor, flygrutter, förkunskaper till högskolekurser
- m.m.

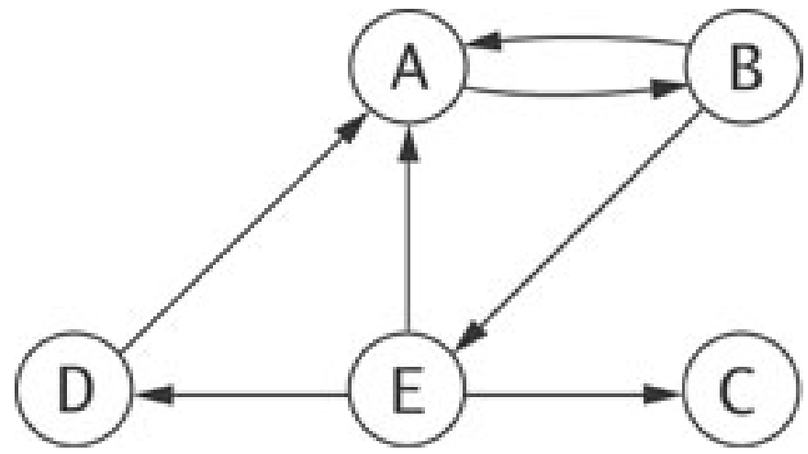
Visuell representation

Noder representeras som punkter eller cirklar

Bågar representeras som linjer mellan noderna



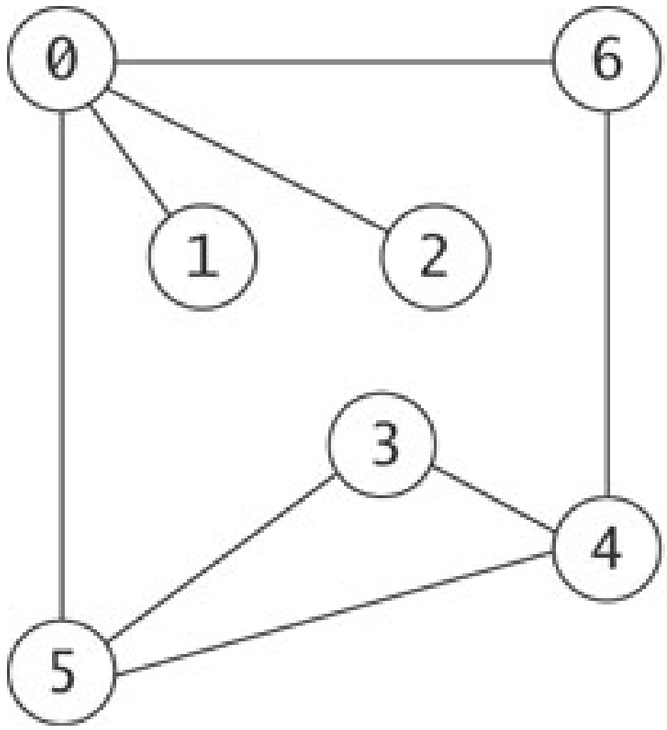
V = {A, B, C, D, E}
E = {(A, B), (A, D),
(C, E), (D, E)}

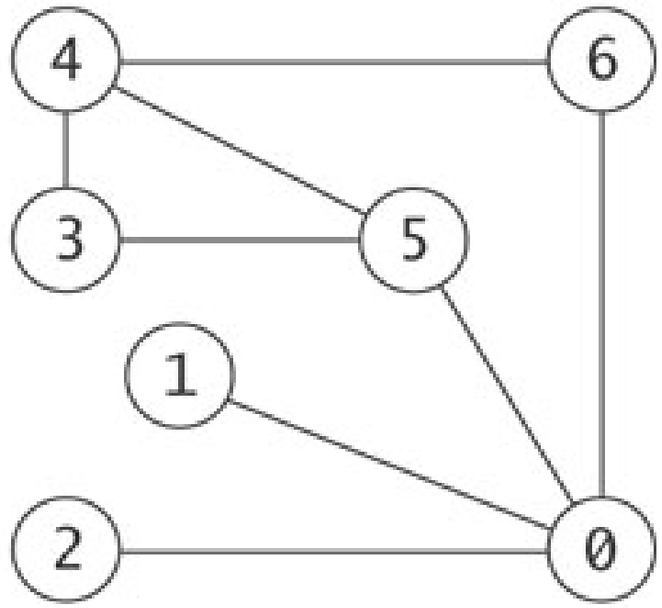


V = {A, B, C, D, E}
E = {(A, B), (B, A), (B, E), (D, A),
(E, A), (E, C), (E, D)}

Layouten är oviktig

Exakt hur vi placerar ut noderna och bågarna är oviktigt





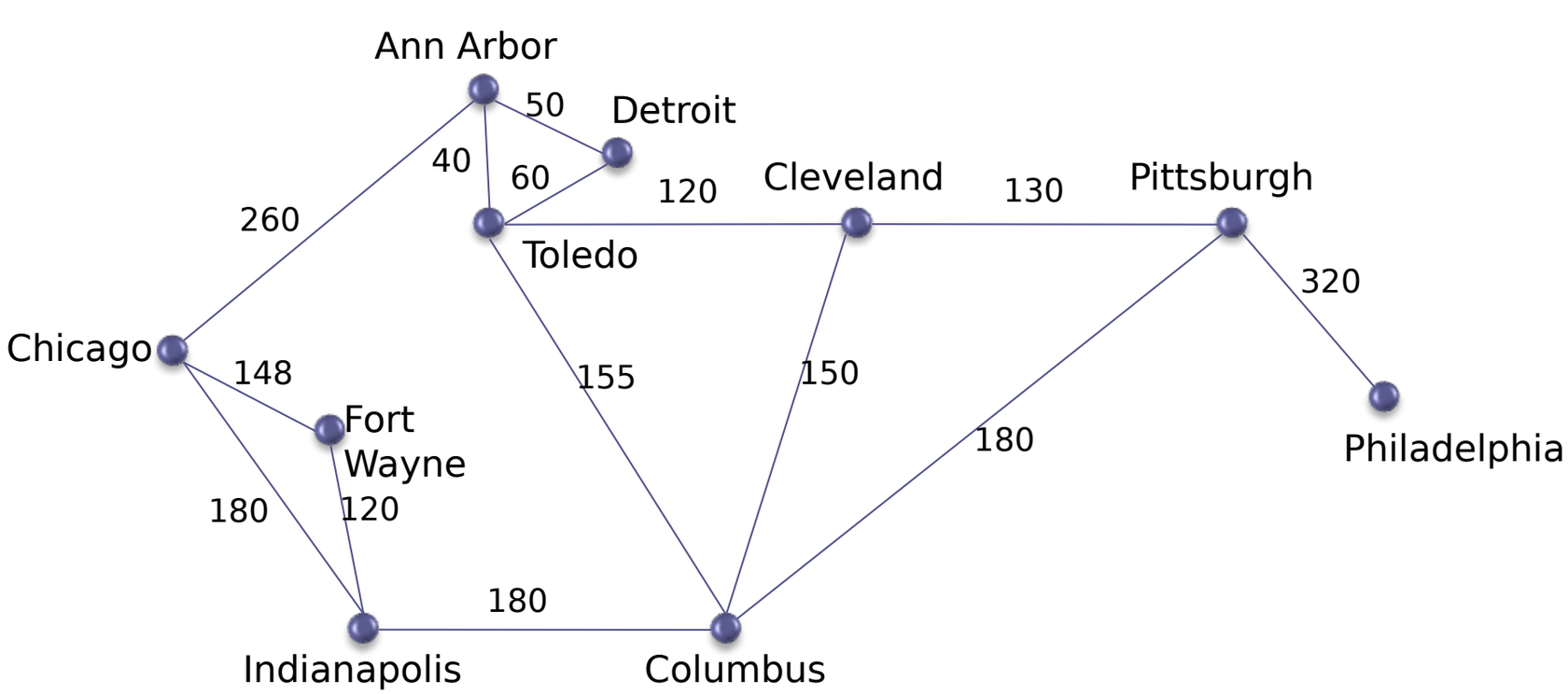
V = {0, 1, 2, 3, 4, 5, 6}
E = {(0, 1), (0, 2), (0, 5), (0, 6), (3, 5), (3, 4), (4, 5), (4, 6)}

Viktade grafer

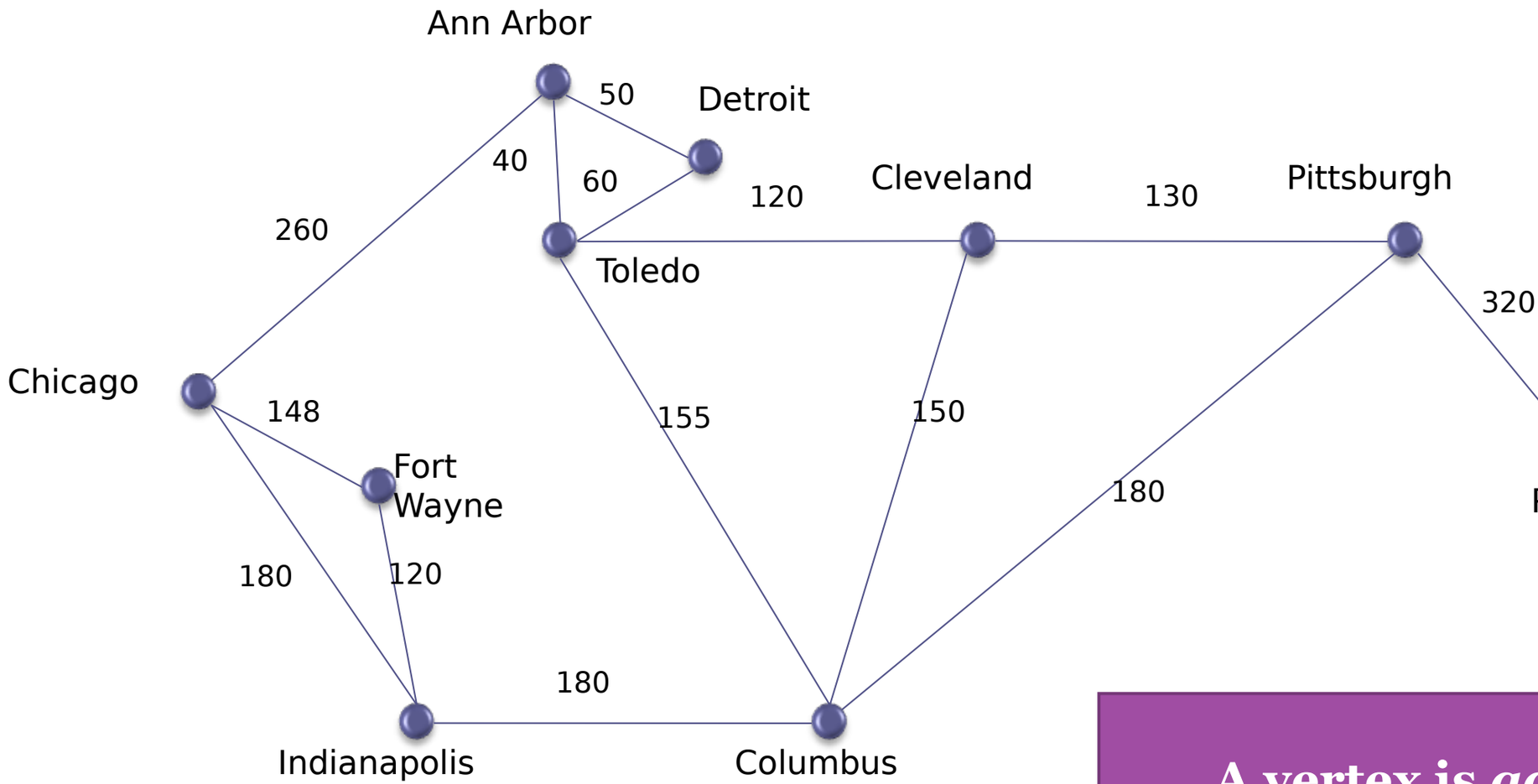
En båge kan ha en *vikt* kopplad till sig

- motsvarande graf blir då en *viktad* graf

En graf kan vara riktad, eller viktad, eller både och

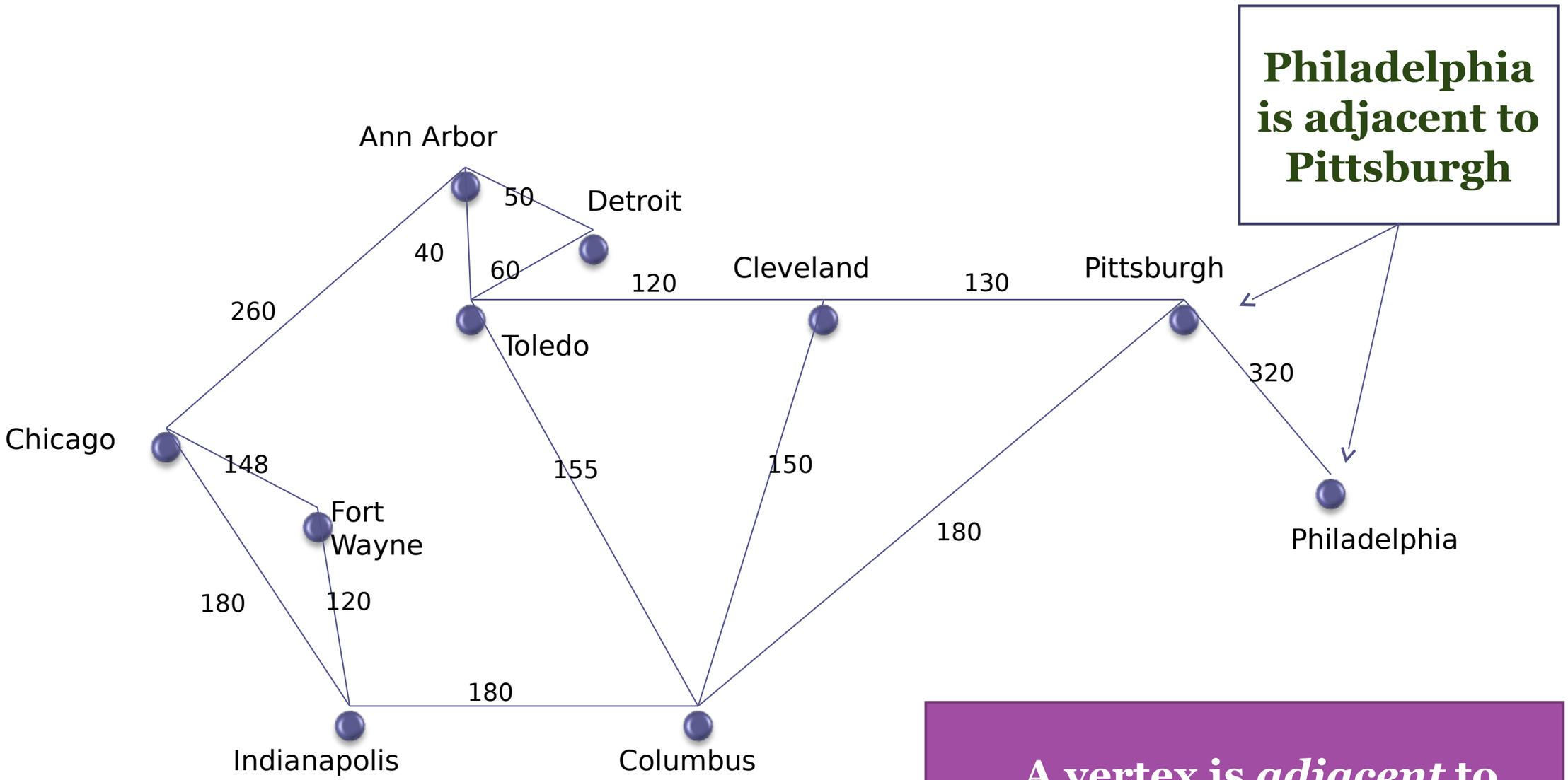


Stigar och cykler



A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

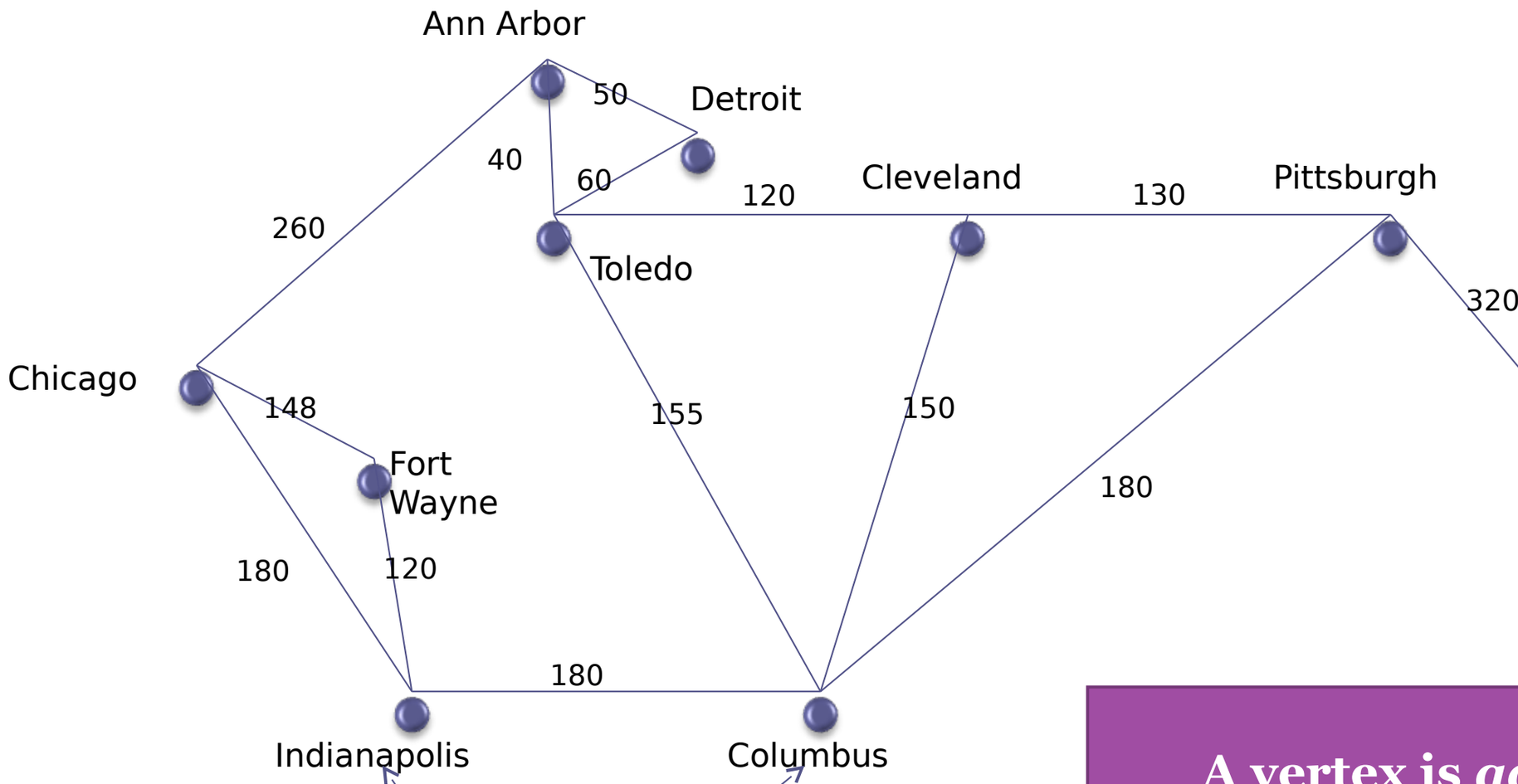
Stigar och cykler



Philadelphia is adjacent to Pittsburgh

A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

Stigar och cykler

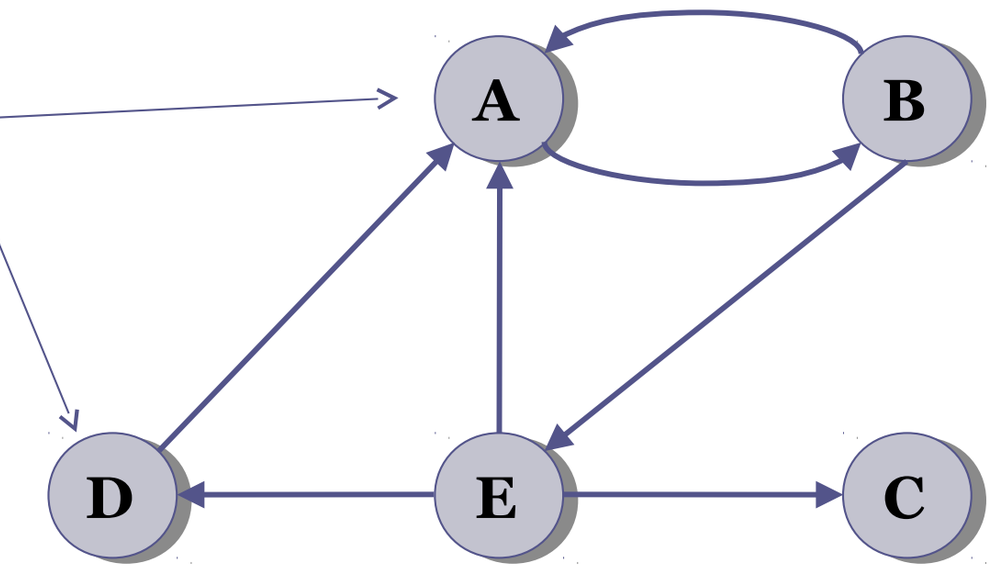


Indianapolis is adjacent to Columbus

A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

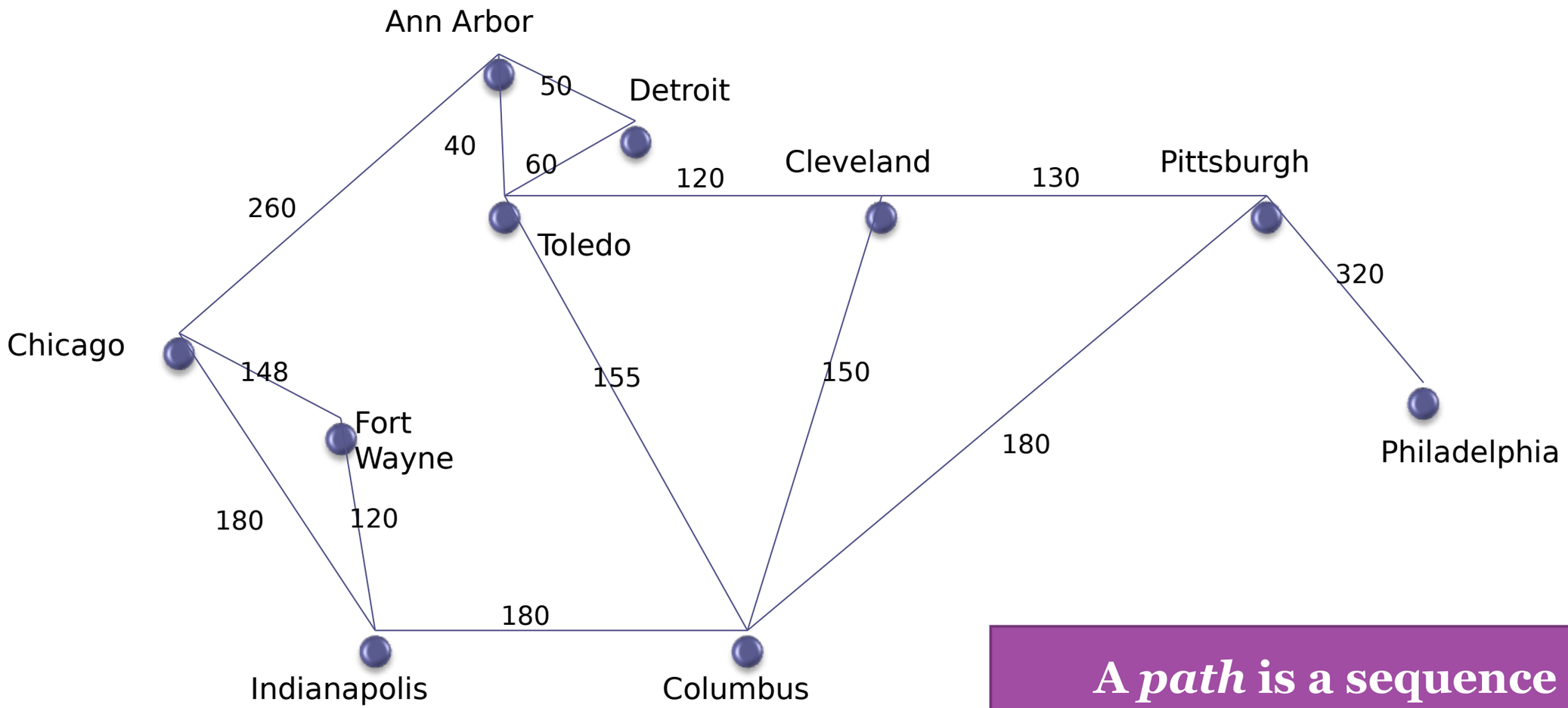
Stigar och cykler

A is adjacent to D, but D is NOT adjacent to A



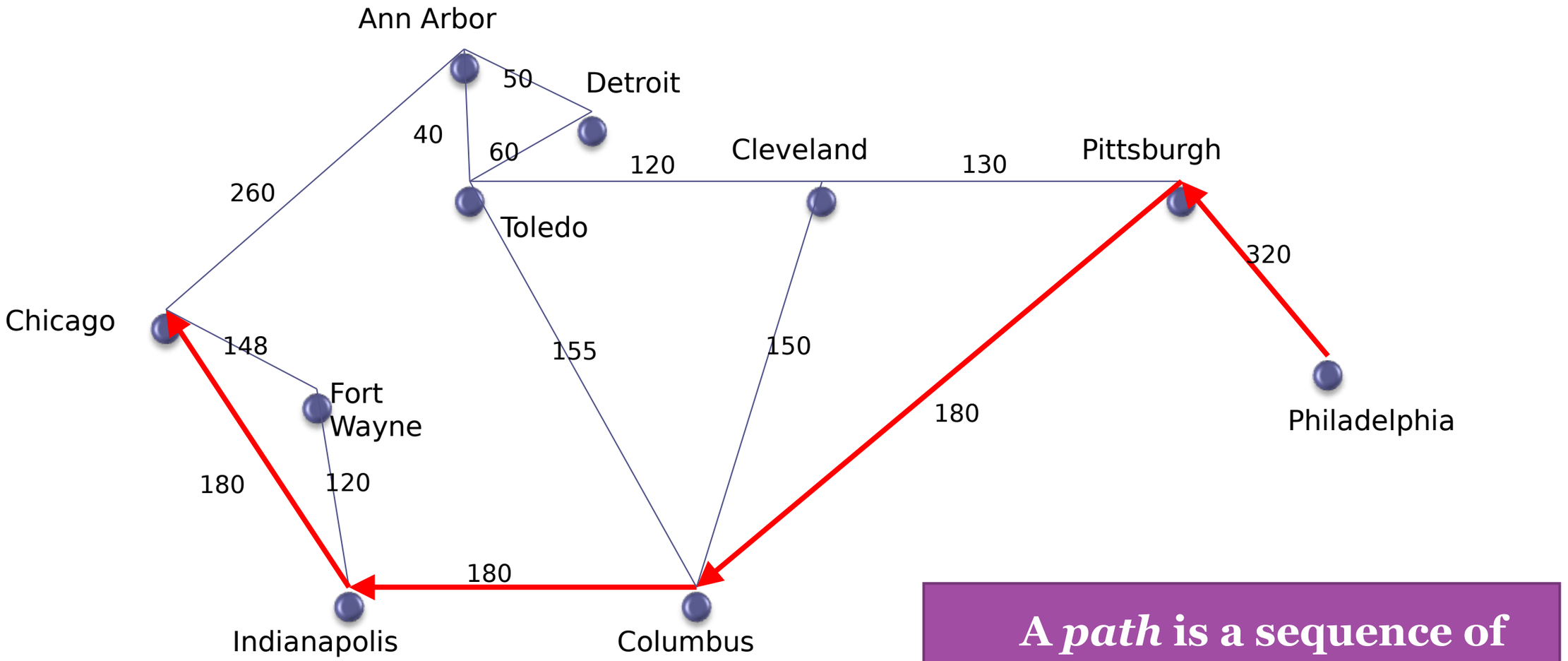
A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

Stigar och cykler



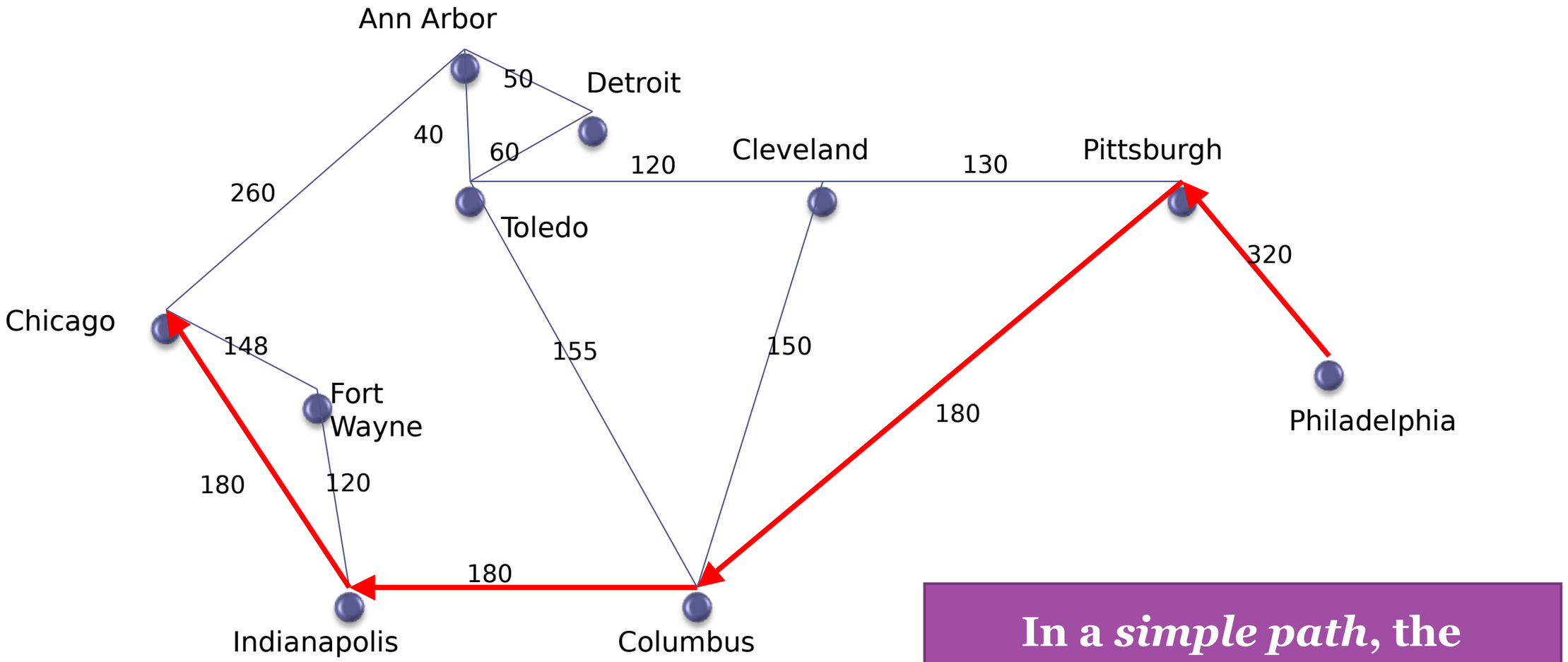
A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor

Stigar och cykler



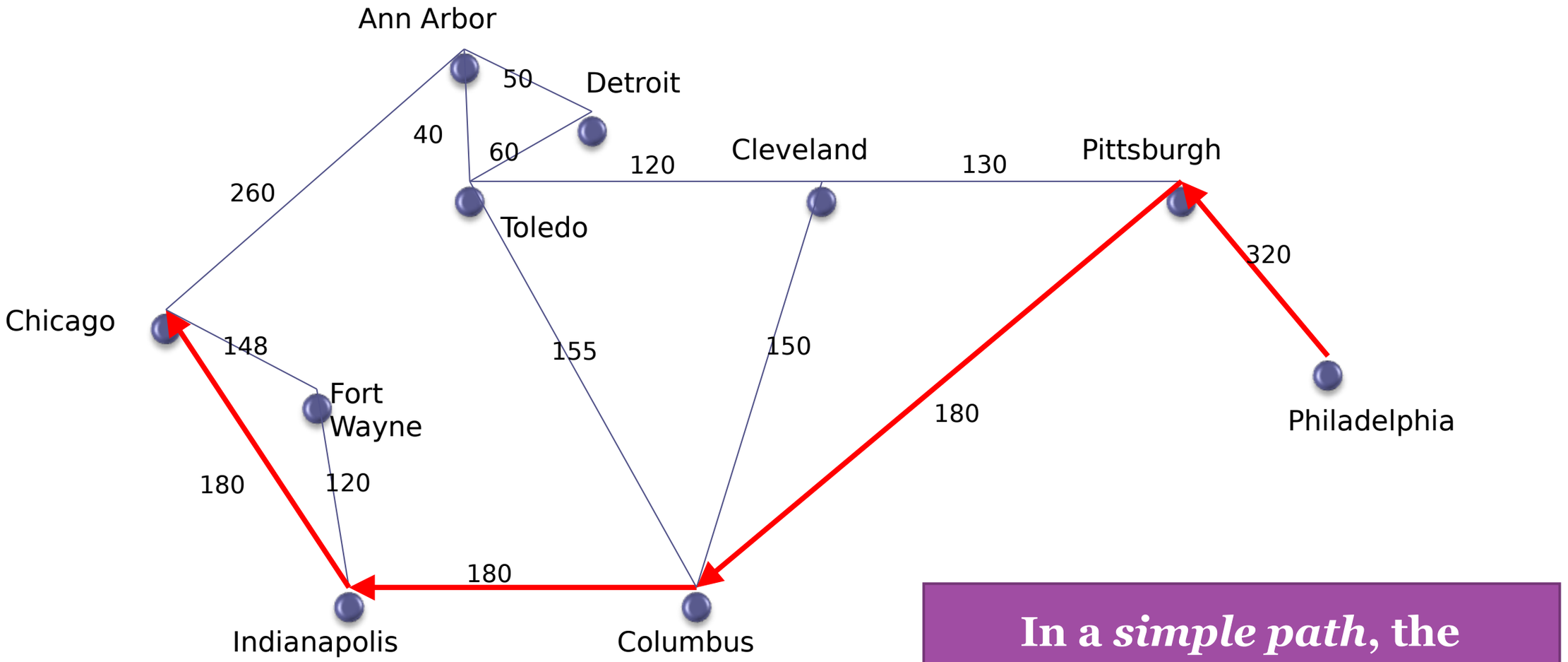
A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor

Stigar och cykler



In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

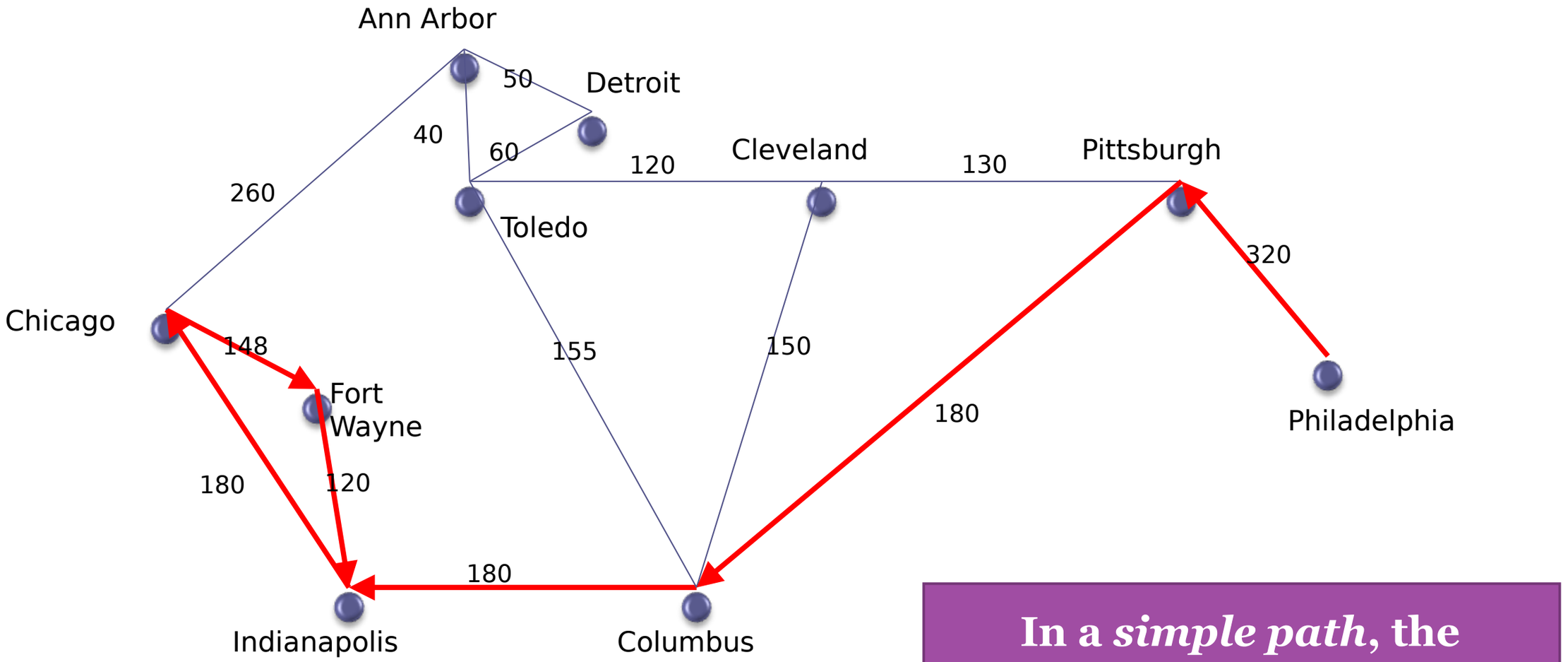
Stigar och cykler



This path is a simple path

In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

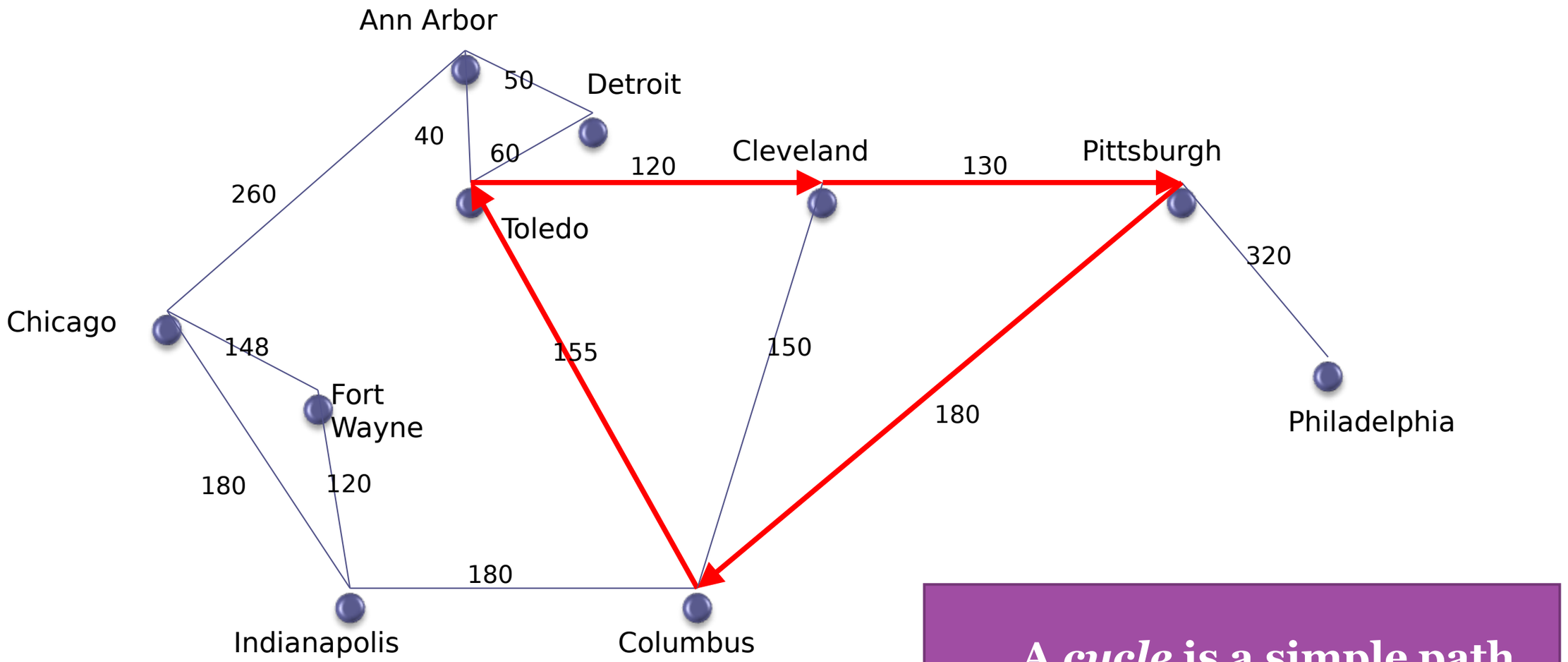
Stigar och cykler



This path is NOT a simple path

In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

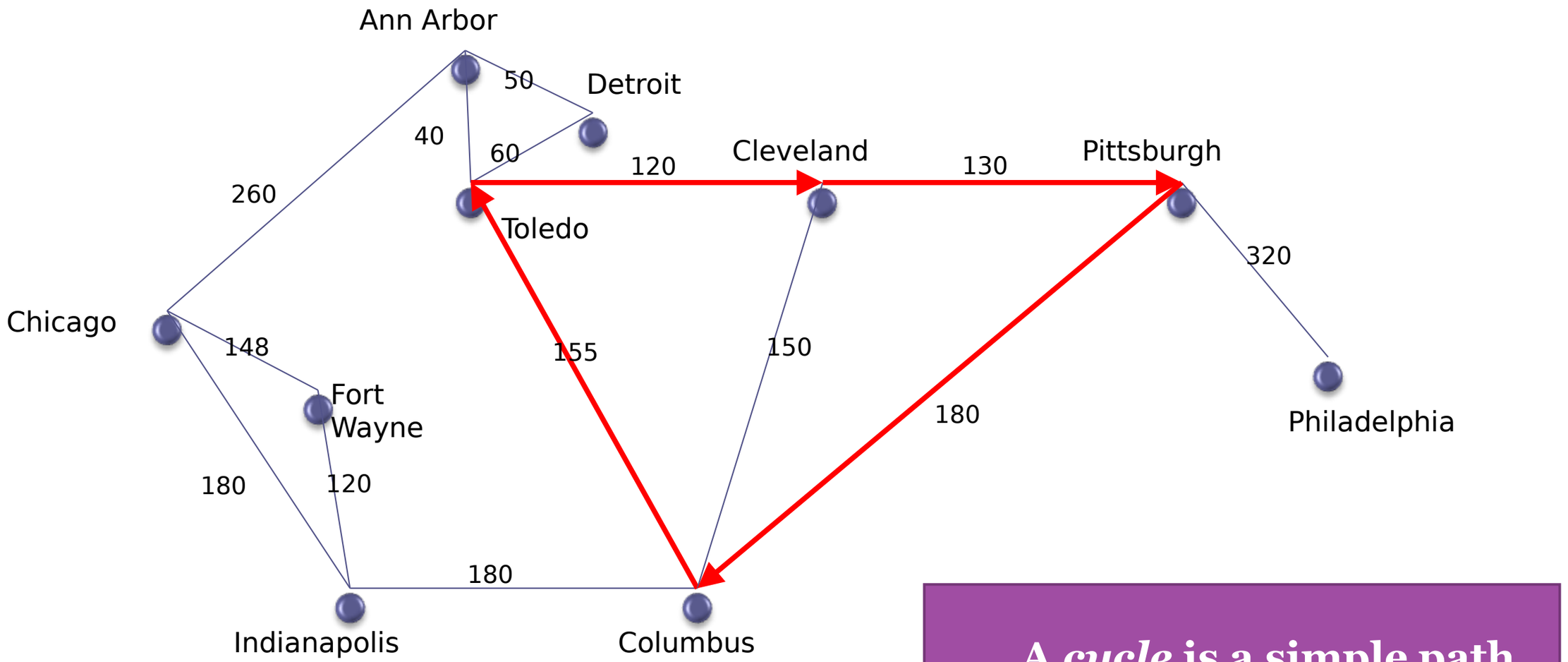
Stigar och cykler



A graph that has a cycle is called *cyclic*
Otherwise it is *acyclic*

A *cycle* is a simple path in which only the first and final vertices are the same

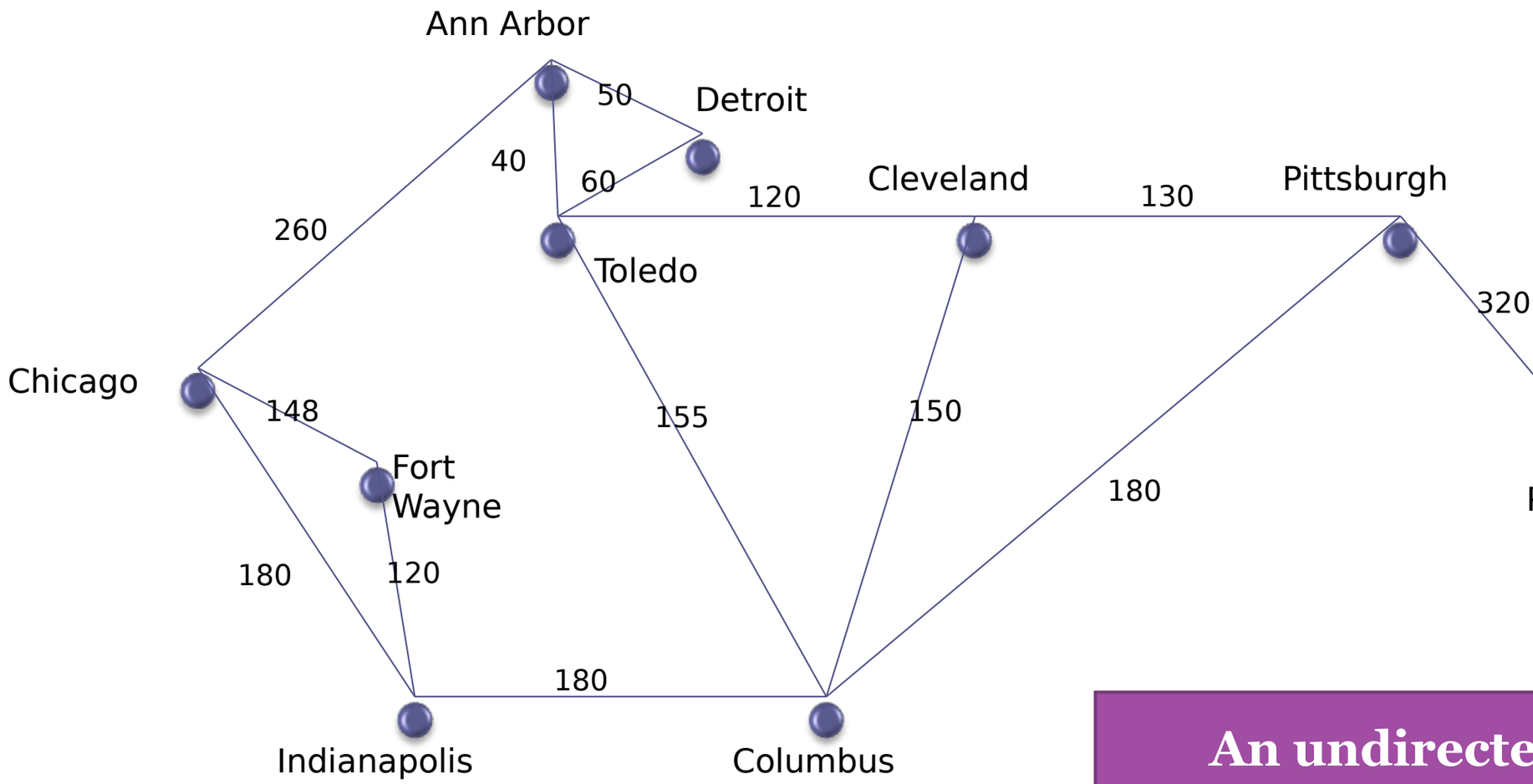
Stigar och cykler



In an undirected graph a cycle must contain at least three distinct vertices
Pittsburgh → Columbus → Pittsburgh is not a cycle

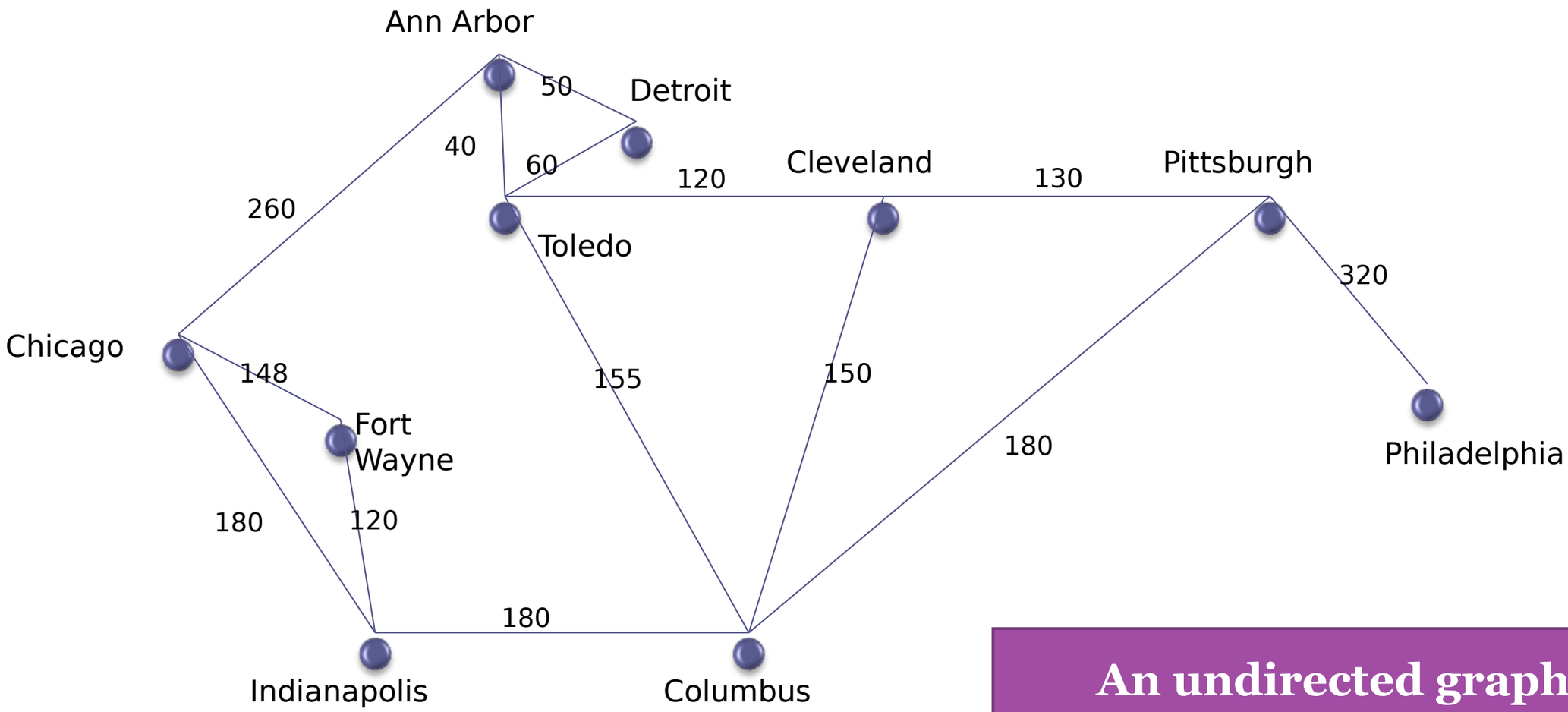
A *cycle* is a simple path in which only the first and final vertices are the same

Stigar och cykler



An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

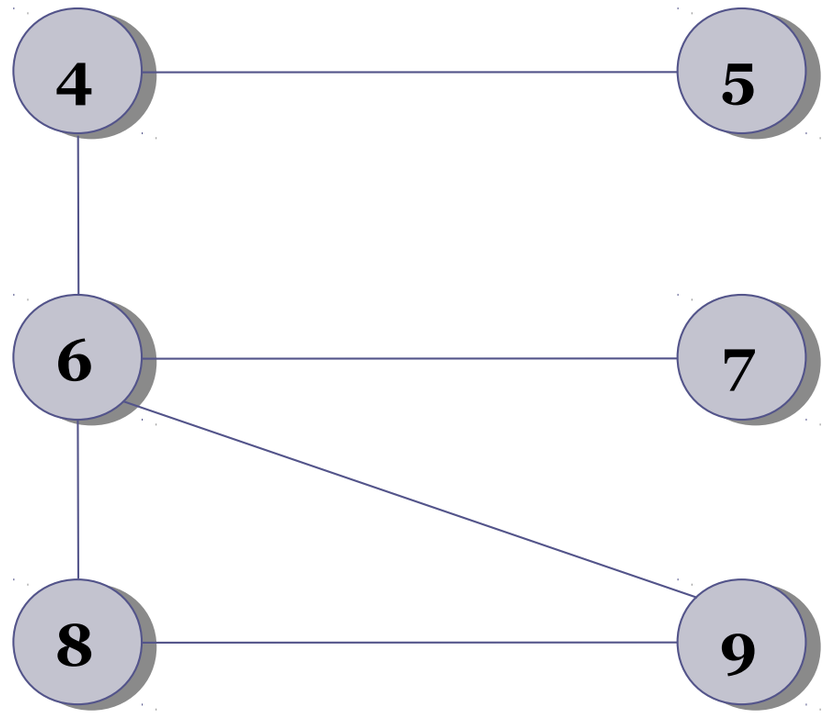
Stigar och cykler



This graph is a connected graph

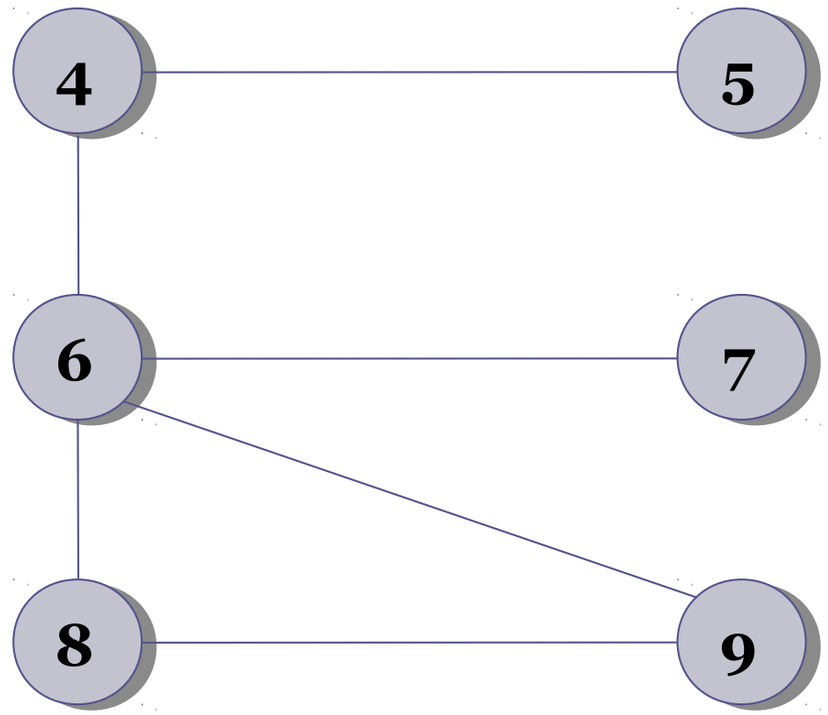
An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

Stigar och cykler



An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

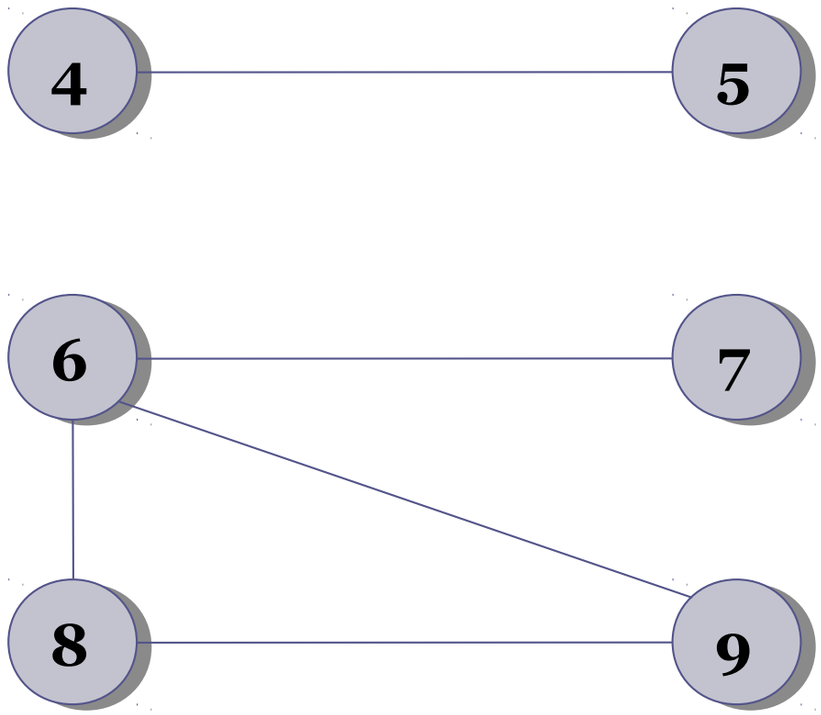
Stigar och cykler



This graph is a connected graph

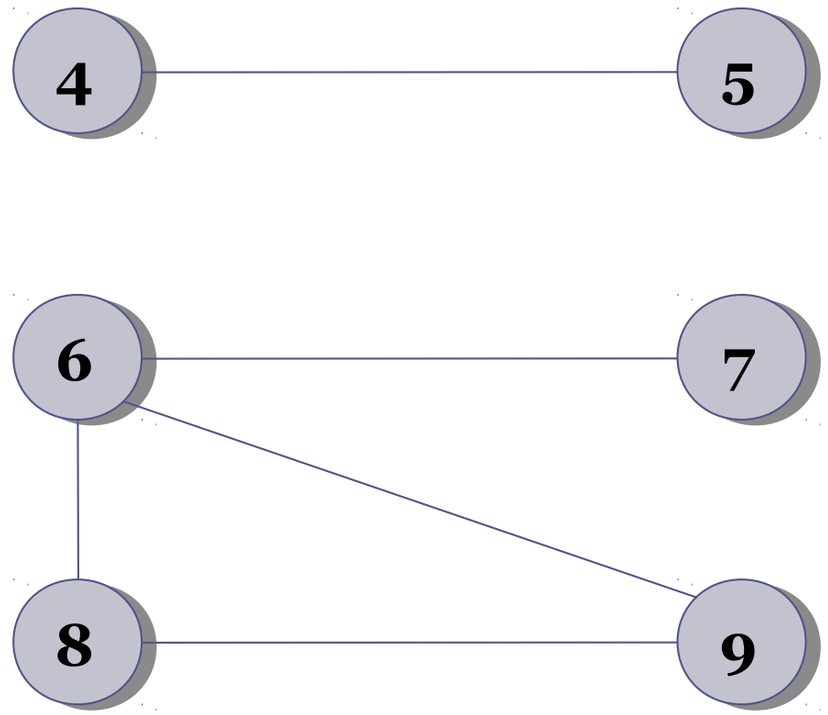
An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

Stigar och cykler



An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

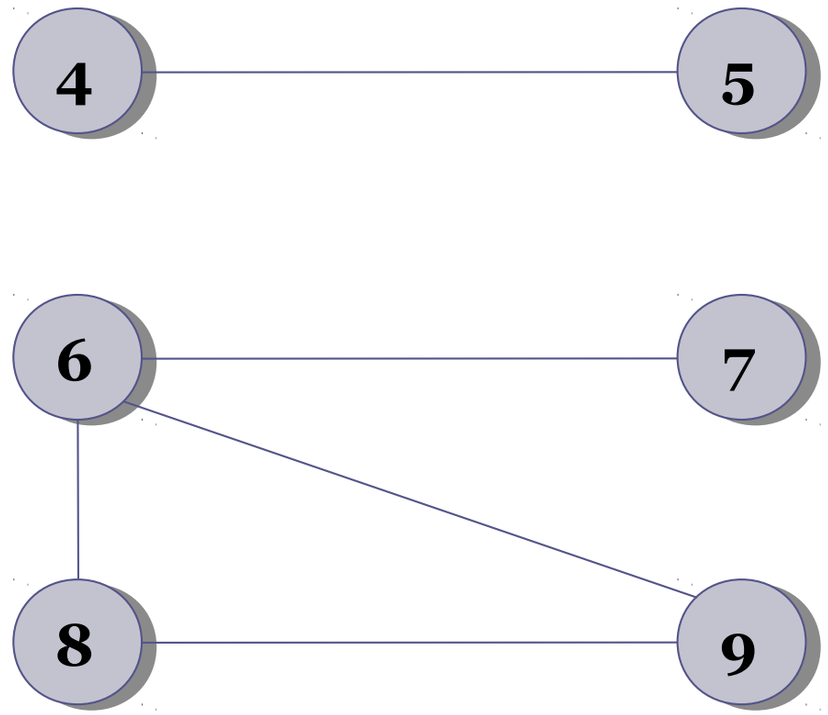
Stigar och cykler



This graph is NOT a connected graph

An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

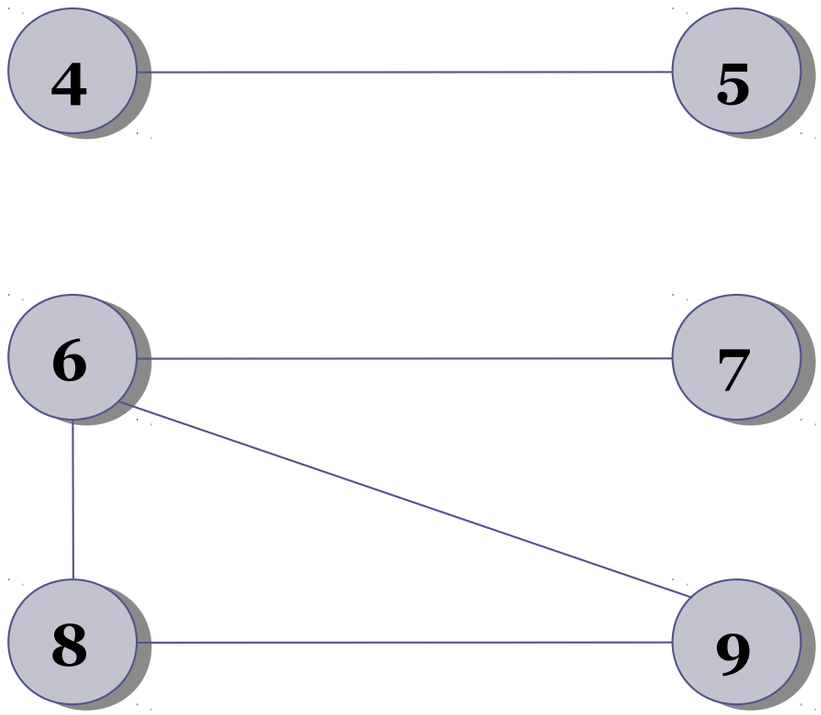
Stigar och cykler



If a graph is not connected, it is considered *unconnected*, but still consists of *connected components*

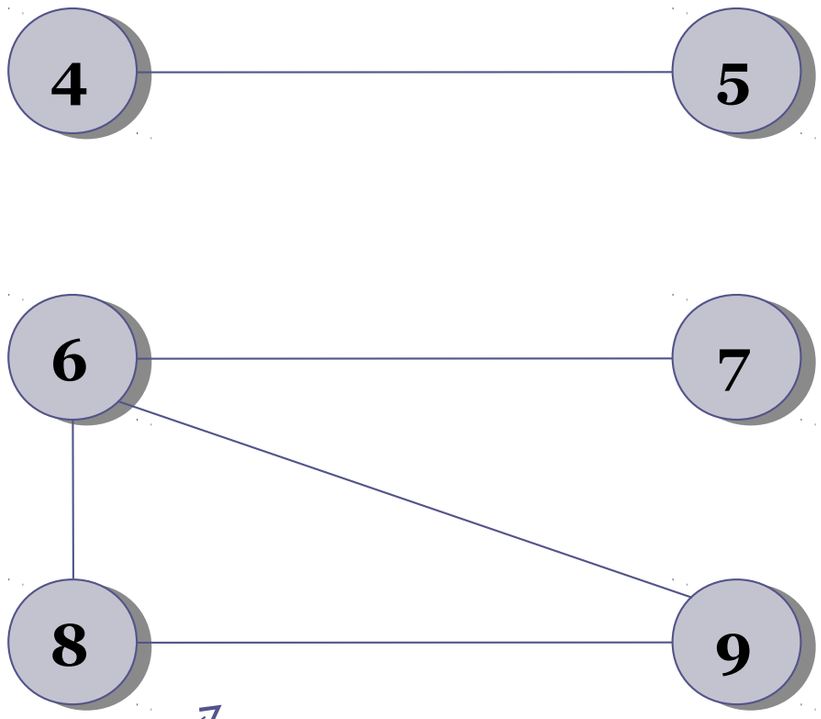
Stigar och cykler

{4, 5} are connected components



If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

Stigar och cykler

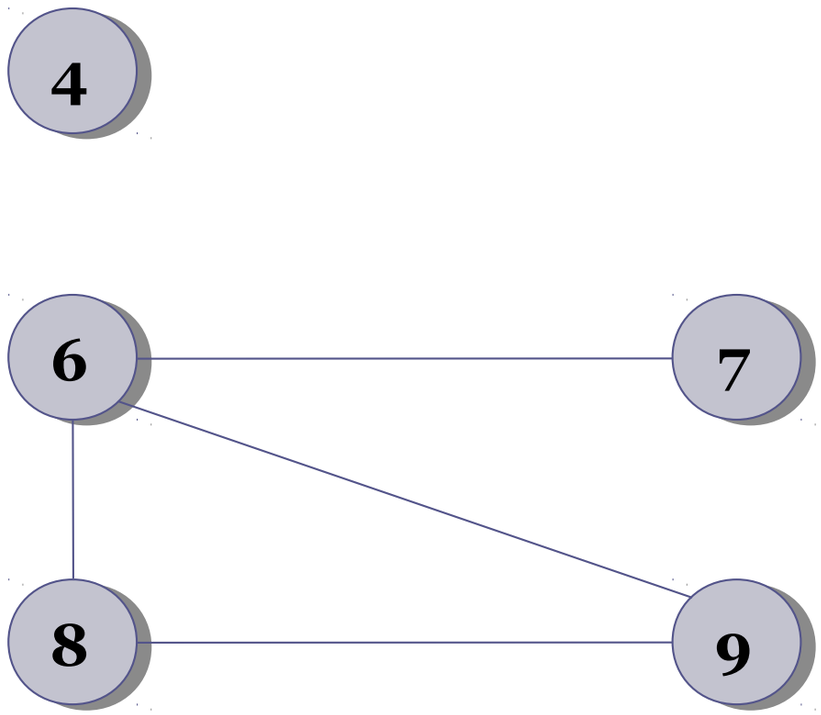


{6, 7, 8, 9} are connected components

If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

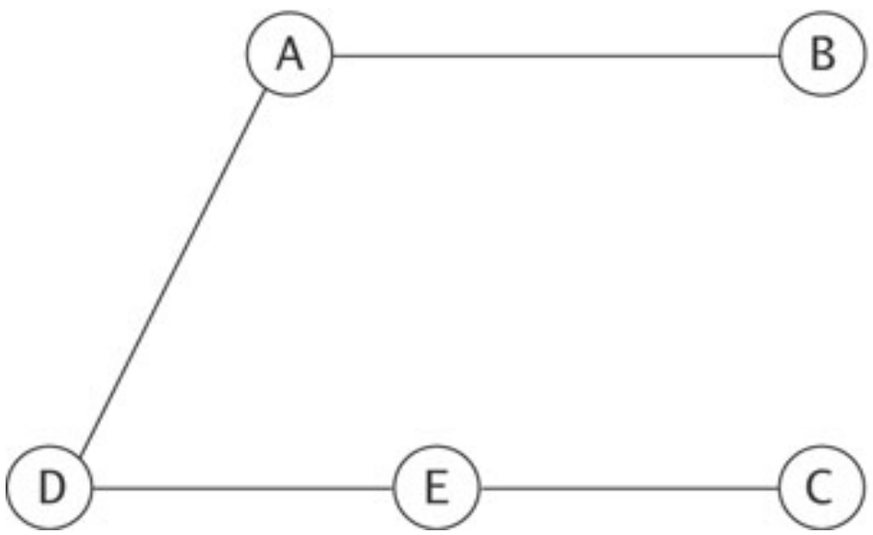
Stigar och cykler

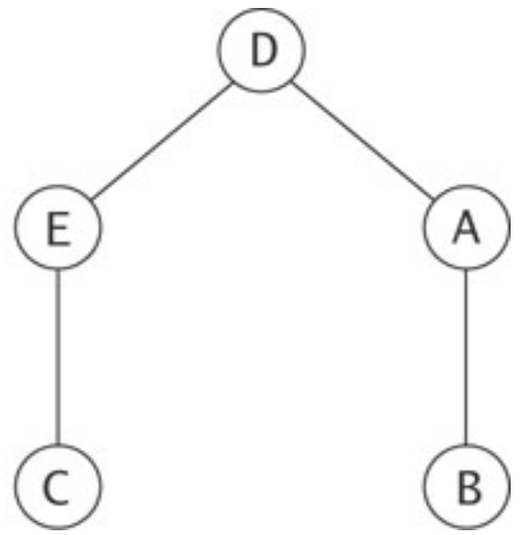
A single vertex with no edge is also considered a connected component



If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

Träd är grafer





Att implementera grafer

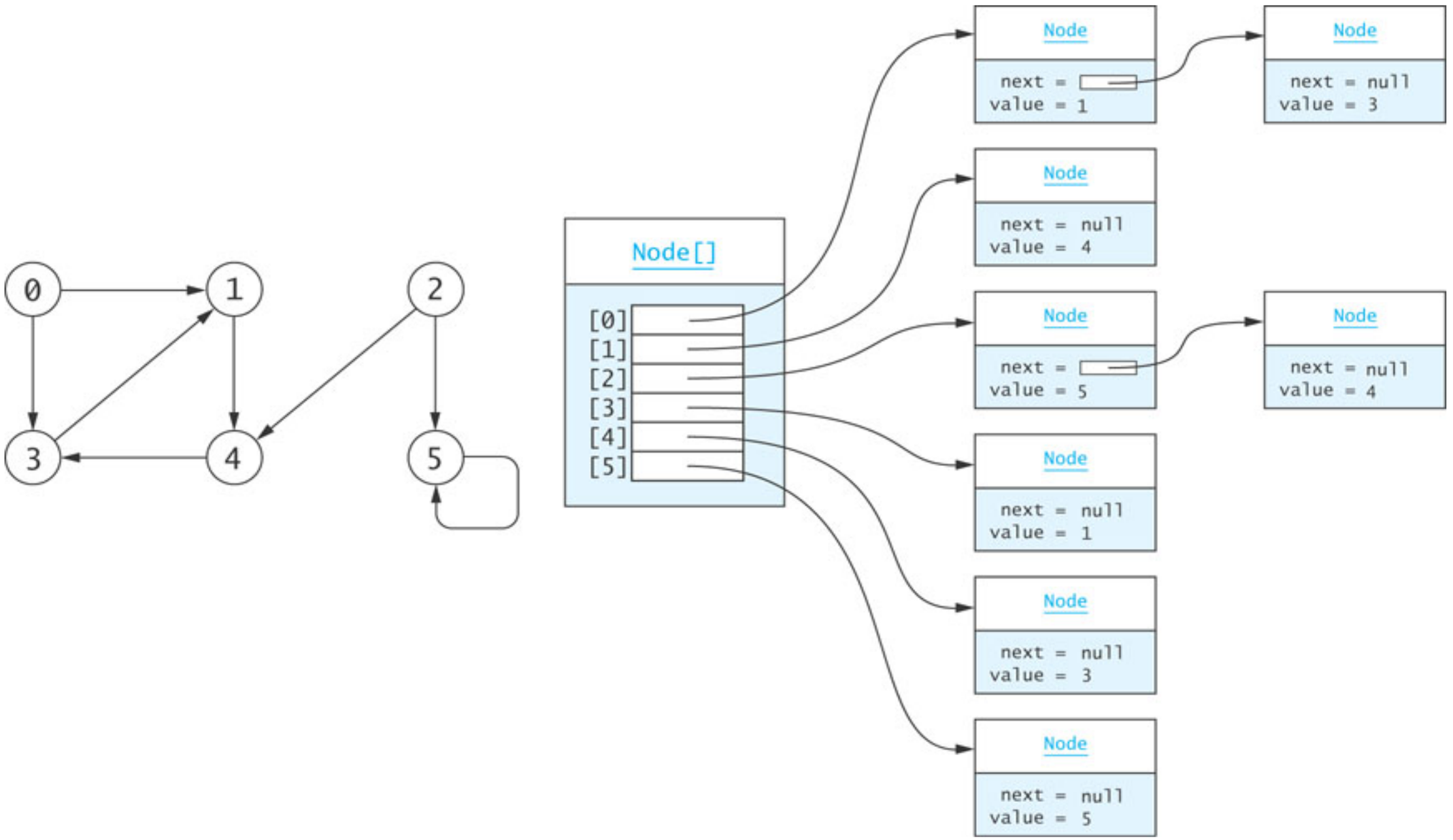
Alternativ 1: Bågarna representeras av en array med |V| listor (kallad ”adjacency lists”), en lista för varje nod

- varje lista innehåller de noder som gränsar till den givna noden
- listan är oordnad

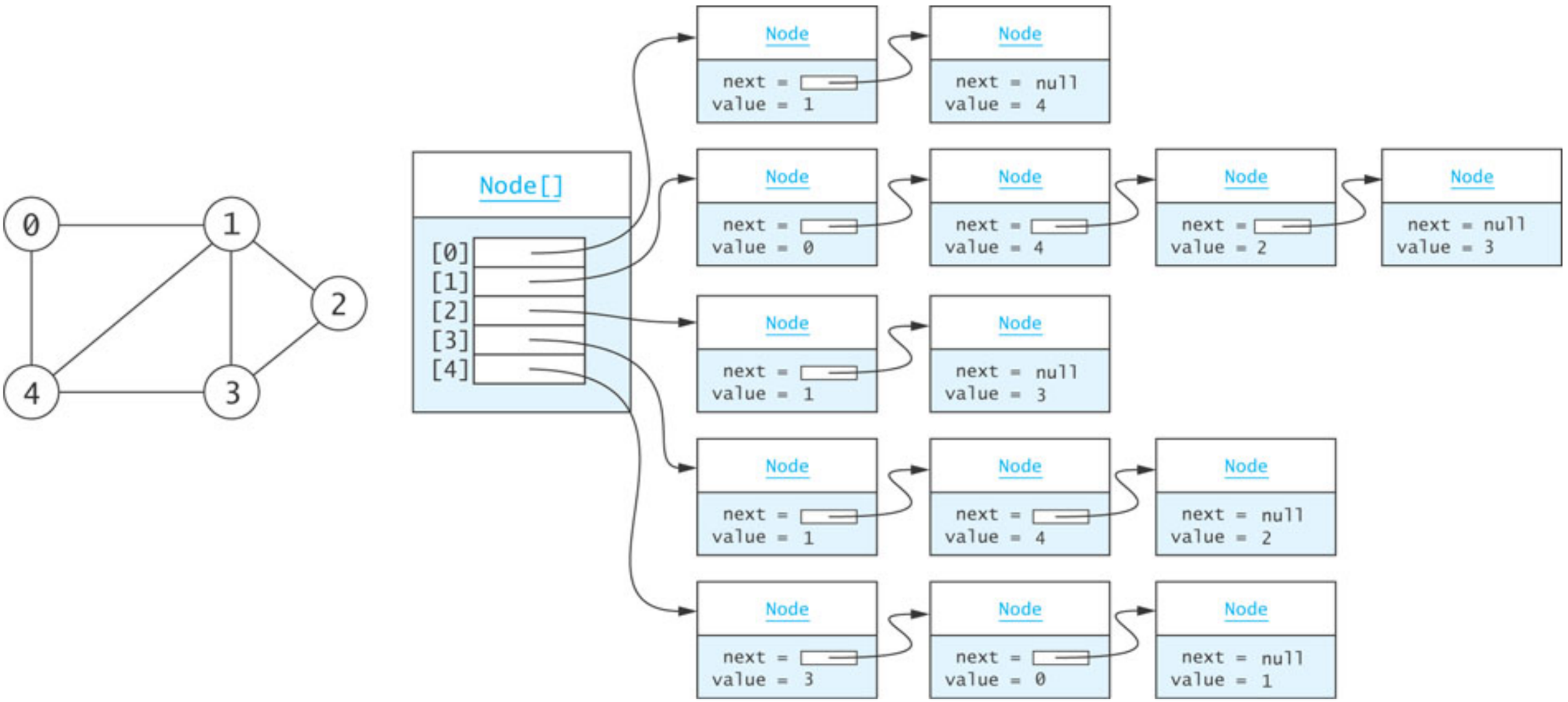
Alternativ 2: Bågarna representeras av en 2-dimensionell array (kallad ”adjacency matrix”), med |V| rader och |V| kolumner

- cellerna kan då innehålla bågens vikt

Adjacency list – riktad graf



Adjacency list – oriktad graf



Adjacency matrix

Vi använder en 2-dimensionell array.

För en oviktad graf, kan cellerna innehålla boolean eller heltal:

- heltalsvärden har ibland en fördel om vi utnyttjar matrismultiplikation, vilket vissa grafalgoritmer gör

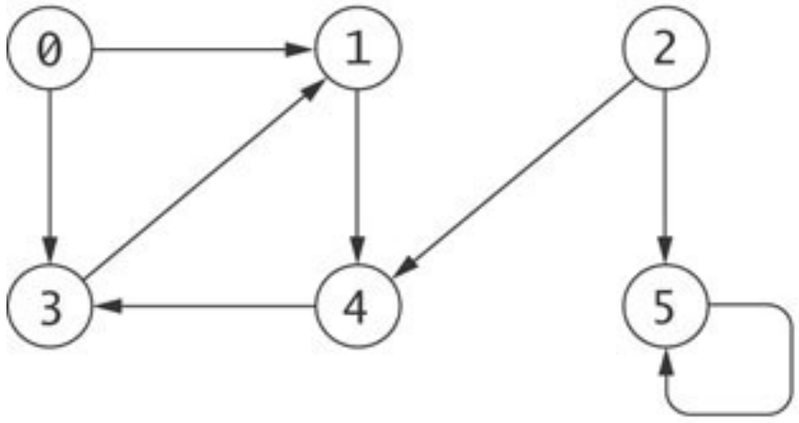
För en viktad graf, kommer cellerna att innhålla vikterna:

- vi använder värdet `Double.POSITIVE_INFINITY` för att representera frånvaron av en båge
- detta för att kunna representera en båge med vikten 0

I en oriktad graf är matrisen symmetrisk över huvuddiagonalen:

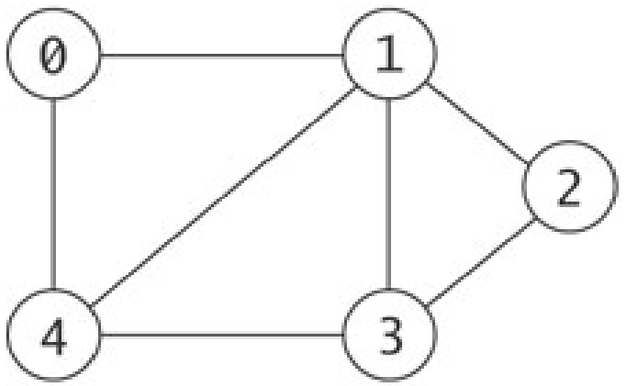
- vi behöver alltså bara använda den nedre vänstra triangeln

Adjacency matrix, oviktad



Column

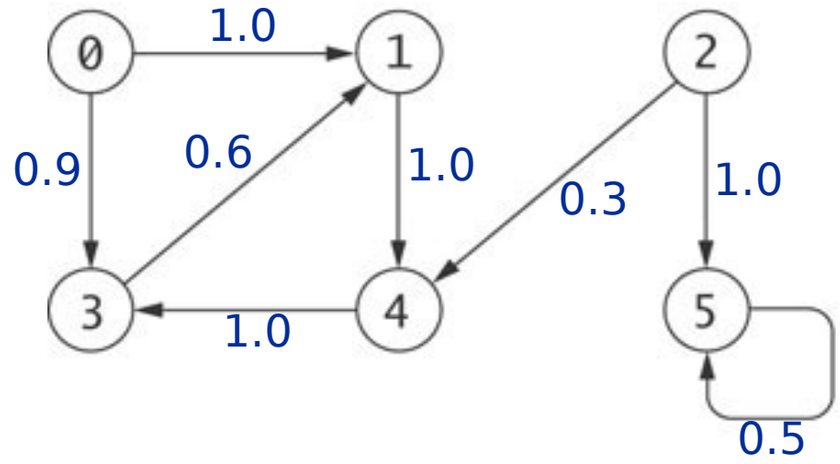
Row	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		1.0		
[1]					1.0	
[2]					1.0	1.0
[3]		1.0				
[4]				1.0		
[5]						1.0



Column

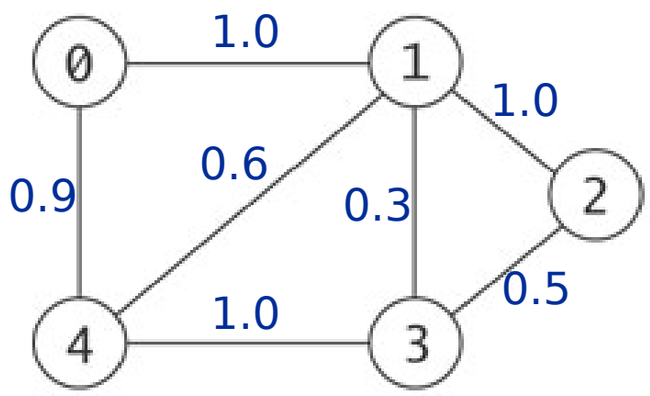
Row	[0]	[1]	[2]	[3]	[4]
[0]		1.0			1.0
[1]	1.0		1.0	1.0	1.0
[2]		1.0		1.0	
[3]		1.0	1.0		1.0
[4]	1.0	1.0		1.0	

Adjacency matrix, viktad



Column

Row	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		0.9		
[1]					1.0	
[2]					0.3	1.0
[3]		0.6				
[4]				1.0		
[5]						0.5



Column

Row	[0]	[1]	[2]	[3]	[4]
[0]		1.0			0.9
[1]	1.0		1.0	0.3	0.6
[2]		1.0		0.5	
[3]		0.3	0.5		1.0
[4]	0.9	0.6		1.0	

Vilken är bäst – lista eller matris

Effektiviteten beror på algoritmen och grafens densitet.

Densiteten är kvoten $|E| / |V|^2$

- i en *tät* (dense) graf är $|E|$ nära $|V|^2$
- i en *gles* (sparse) graf är $|E|$ mycket mindre än $|V|^2$

Vi kan anta att

- $O(|E|) = O(|V|^2)$ för en tät graf
- $O(|E|) = O(|V|)$ för en gles graf

Vilken är bäst?

Många grafalgoritmer är på formen

```
for each vertex u in the graph:
        for each vertex v adjacent to u:
        do something with edge (u, v)
```

I en adjacency list, så kommer vi att gå igenom varje båge exakt en gång, vilket ger en komplexitet på O(|E|).

I en adjacency matrix, så måste vi även pröva alla par (*u*, *v*) som inte har någon båge. Steg 1 och 2 är O(|V|) vardera, vilket ger en komplexitet på $O(|V|^2)$.

- om grafen är tät så är $O(|E|) = O(|V|^2)$, och båda representationerna är lika effektiva
- men om grafen är gles så är O(|E|) = O(|V|), och list-representationen är effektivare

Vilken är bäst?

En del grafalgoritmer är på formen

```
for each vertex u in some subset of the vertices:
        for each vertex v in some subset of the vertices:
            if (u, v) is an edge:
                do something with edge (u, v)
```

I en adjacency matrix, så är steg 1 och 2 O(|V|) var, och steg 3 är O(1). Hela algoritmen blir alltså $O(|V|^2)$.

I en adjacency list, så är steg 1 O(|V|), medan kombinationen av steg 2 och 3 är O(|E|). Hela algoritmen blir alltså O(|V| |E|).

- om grafen är gles så är O(|E|) = O(|V|), och båda representationerna är lika effektiva
- men i en tät graf är $O(|E|) = O(|V|^2)$, vilket gör att list-representationen blir $O(|V|^3)$, och matris-representationen är effektivare

Vilken är bäst?

Alltså, för tidskomplexiteten gäller:

- om grafen är tät så är adjacency matrix bättre
- om grafen är gles så är adjacency list bättre

Hur är det med minnet?

- en adjacency matrix behöver allokera plats för $|V|^2$ celler
- en adjacency list behöver bara plats för |E| Edge-objekt
- men varje Edge innehåller pekare till *source*, *destination*, *weight* och *nästa båge* i listan
- alltså behöver en adjacency list plats för $4 \cdot |E|$ värden
- dvs, om grafen är ca 25% full så tar representationerna ungefär lika mycket minne

Grafer, traversering

Traversering av grafer

De flesta grafalgoritmer innebär att besöka varje nod i någon systematisk ordning

- precis som med träd så finns det olika sätt att göra detta på

De två vanligaste metoderna är:

- bredden-först-sökning
- djupet-först-sökning

BFS: Bredden-först-sökning

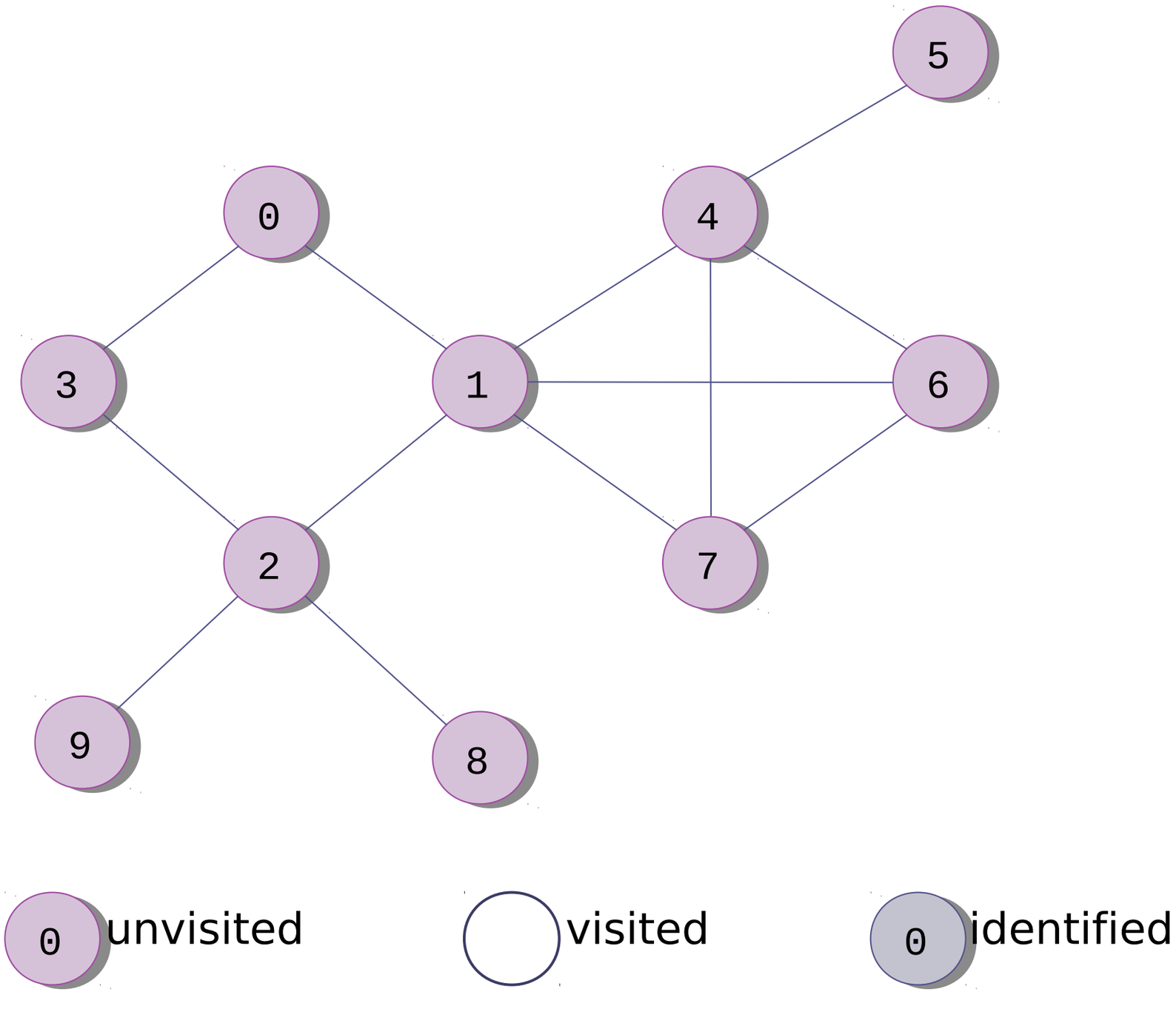
Vid bredden-först så besöker vi noderna i följande ordning:

- besök startnoden först
- sedan alla angränsande noder
- sedan alla noder som kan nås via två bågar
- sedan alla noder som kan nås via tre bågar
- och så vidare

Vi besöker alltså alla noder som kan nås i k steg, innan vi besöker de noder som kan nås i k+1 steg.

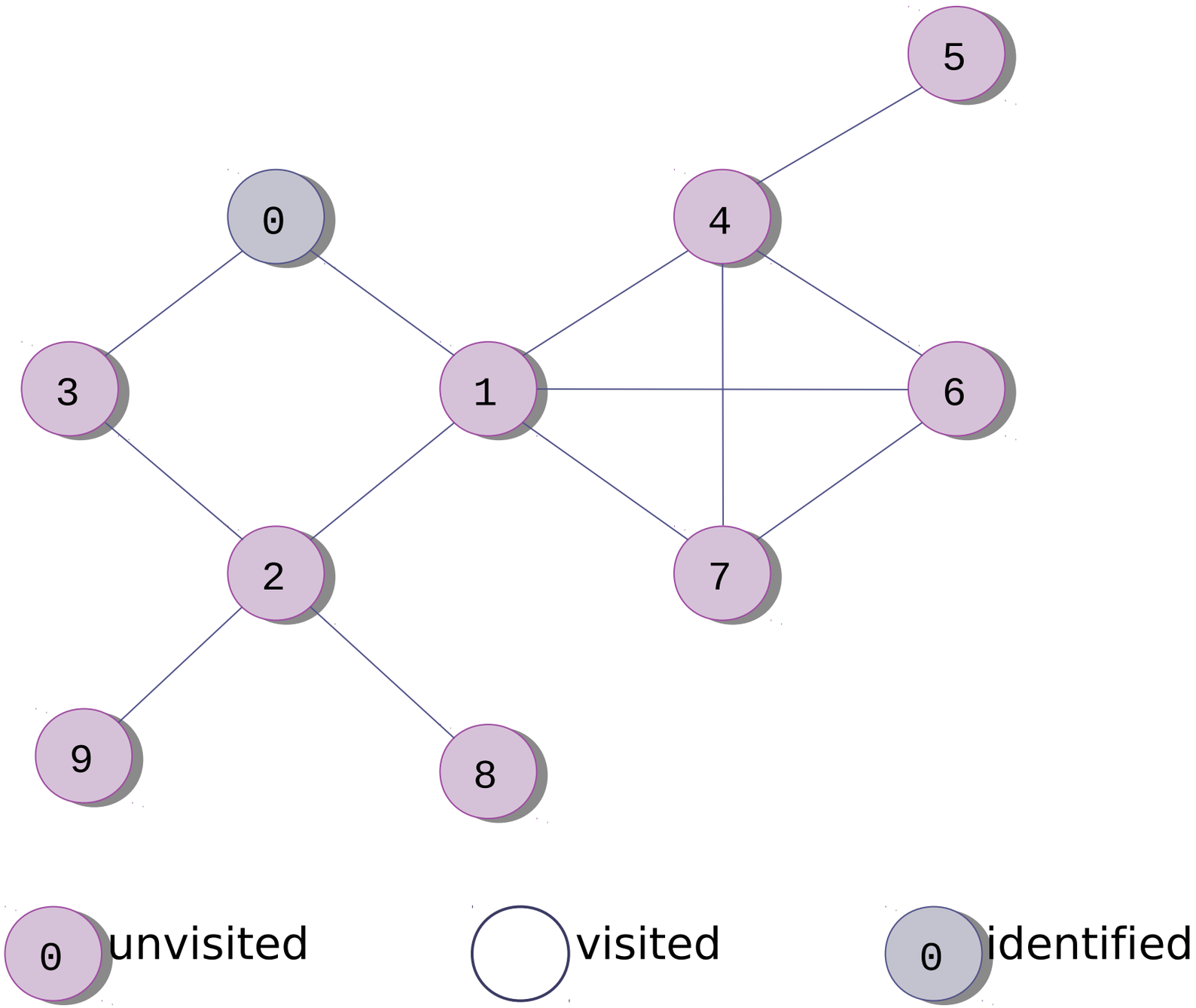
Eftersom ingen nod är speciell så antar vi för enkelhets skull att nod nr 0 är startnoden.

Example of a Breadth-First Search



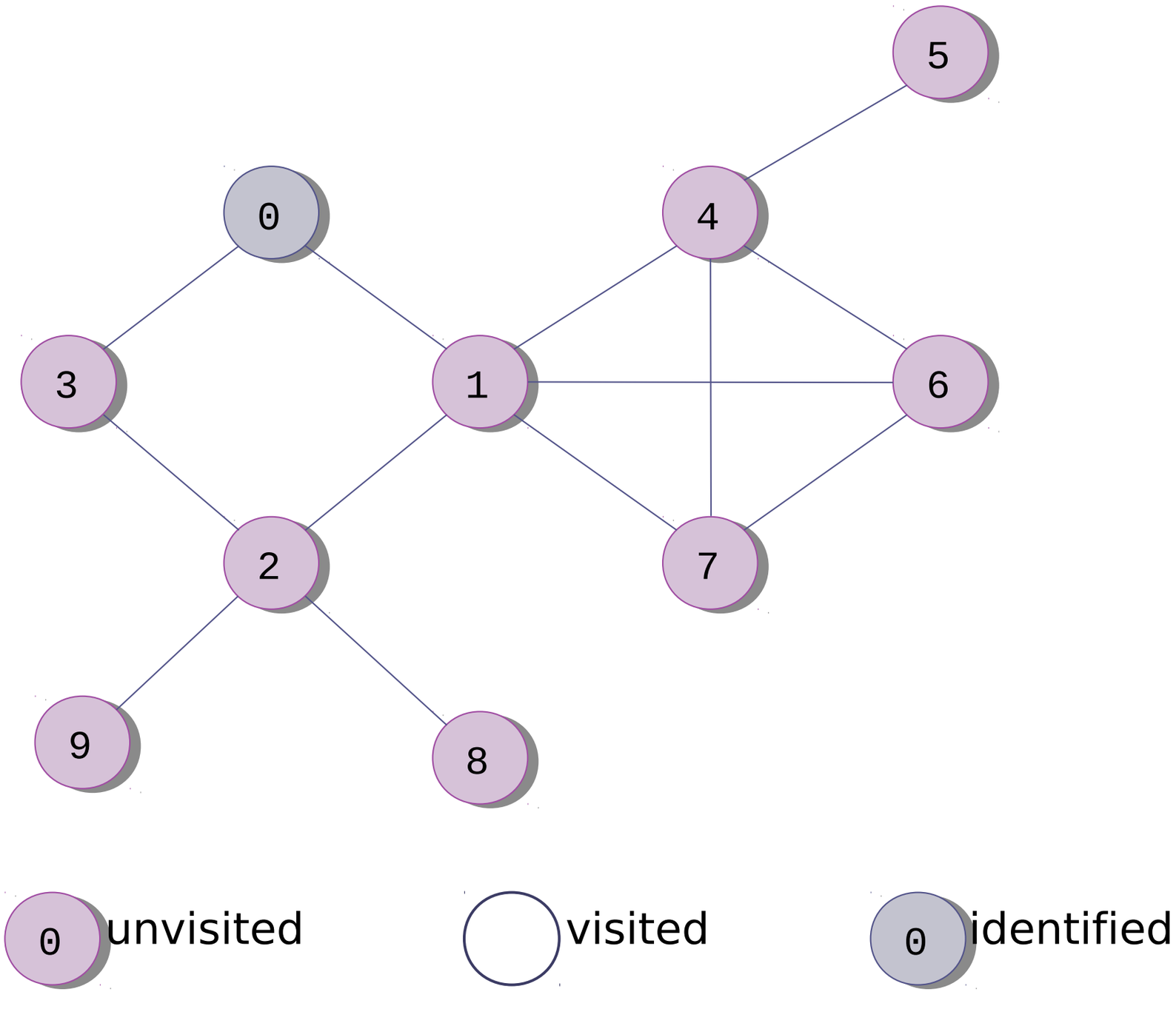
Example of a Breadth-First Search (cont.)

Identify the start node



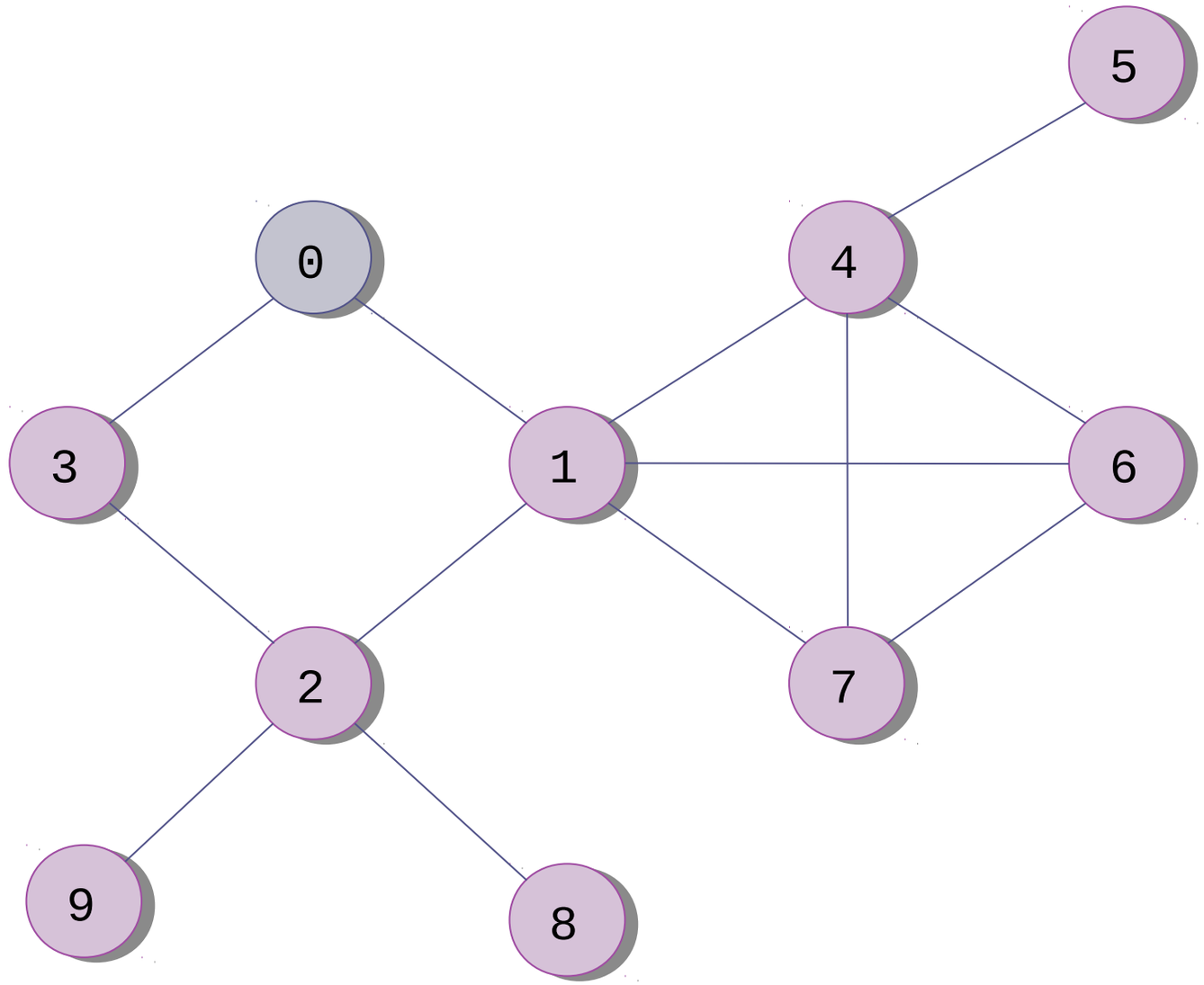
Example of a Breadth-First Search (cont.)

While visiting it, we can identify its adjacent nodes



Example of a Breadth-First Search (cont.)

We identify its adjacent nodes and add them to a queue of identified nodes



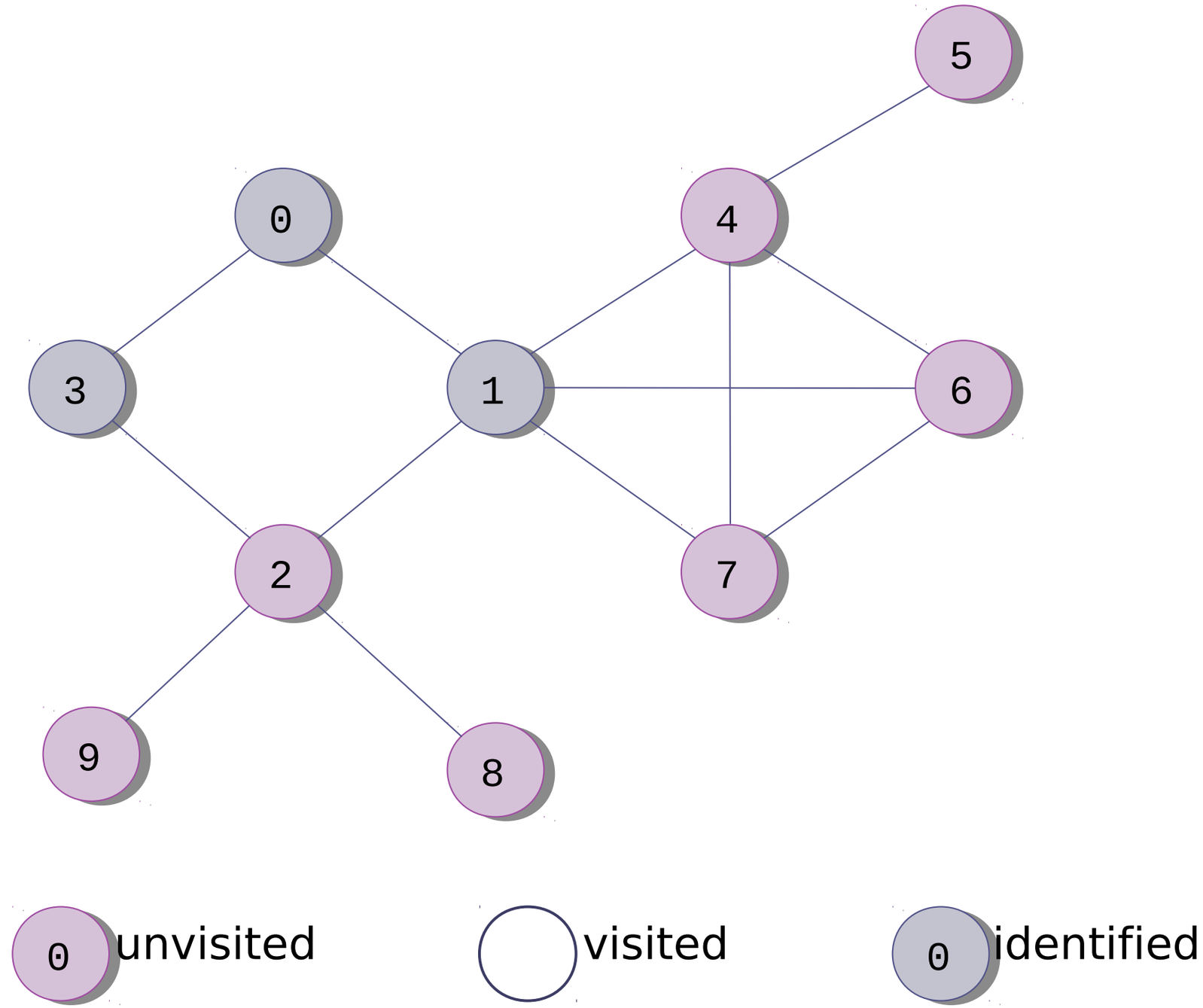
Visit sequence:
0

Example of a Breadth-First Search (cont.)

We identify its adjacent nodes and add them to a queue of identified nodes

Queue:
1, 3

Visit sequence:
0

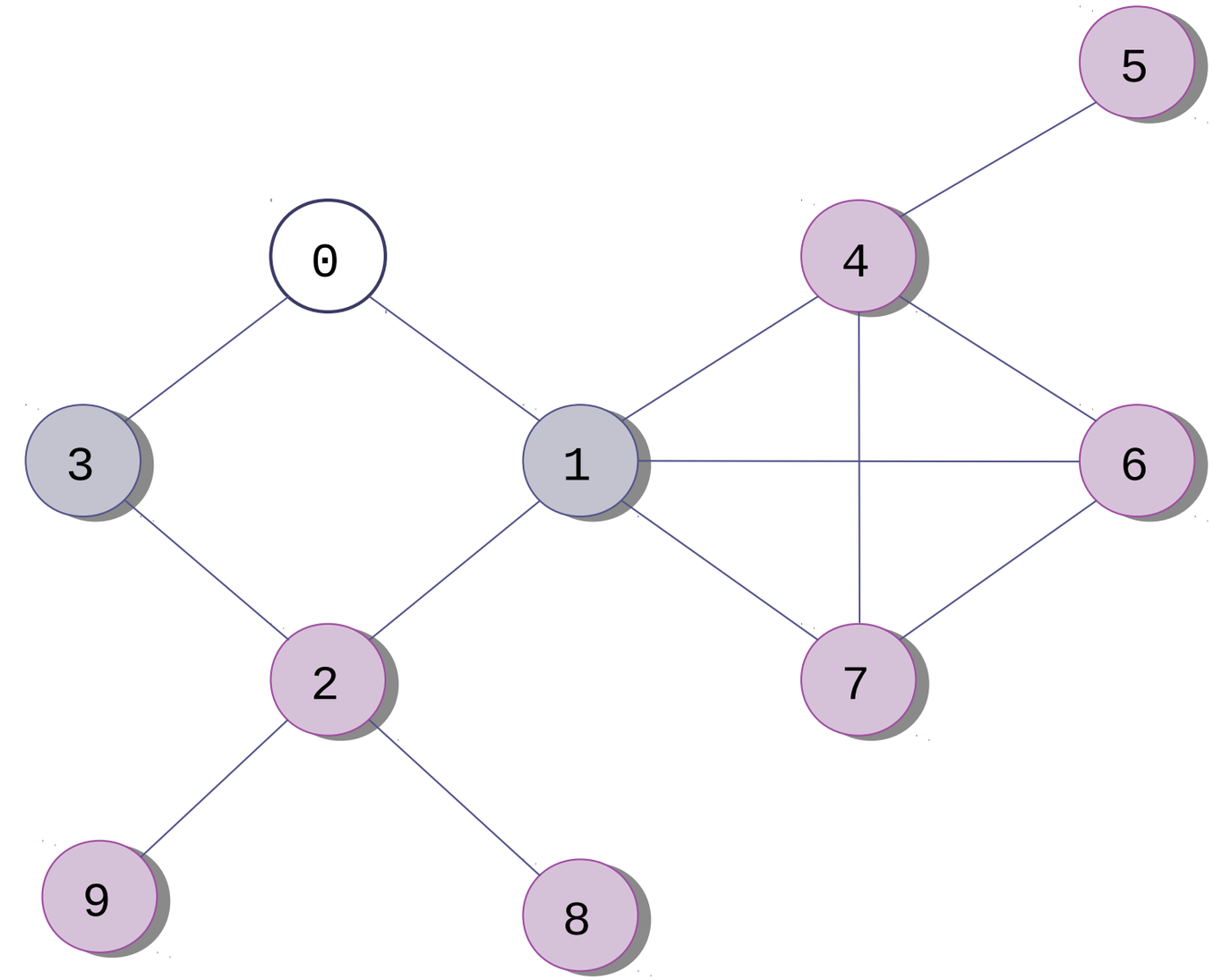


Example of a Breadth-First Search (cont.)

We color the node as visited

Queue:
1, 3

Visit sequence:
0

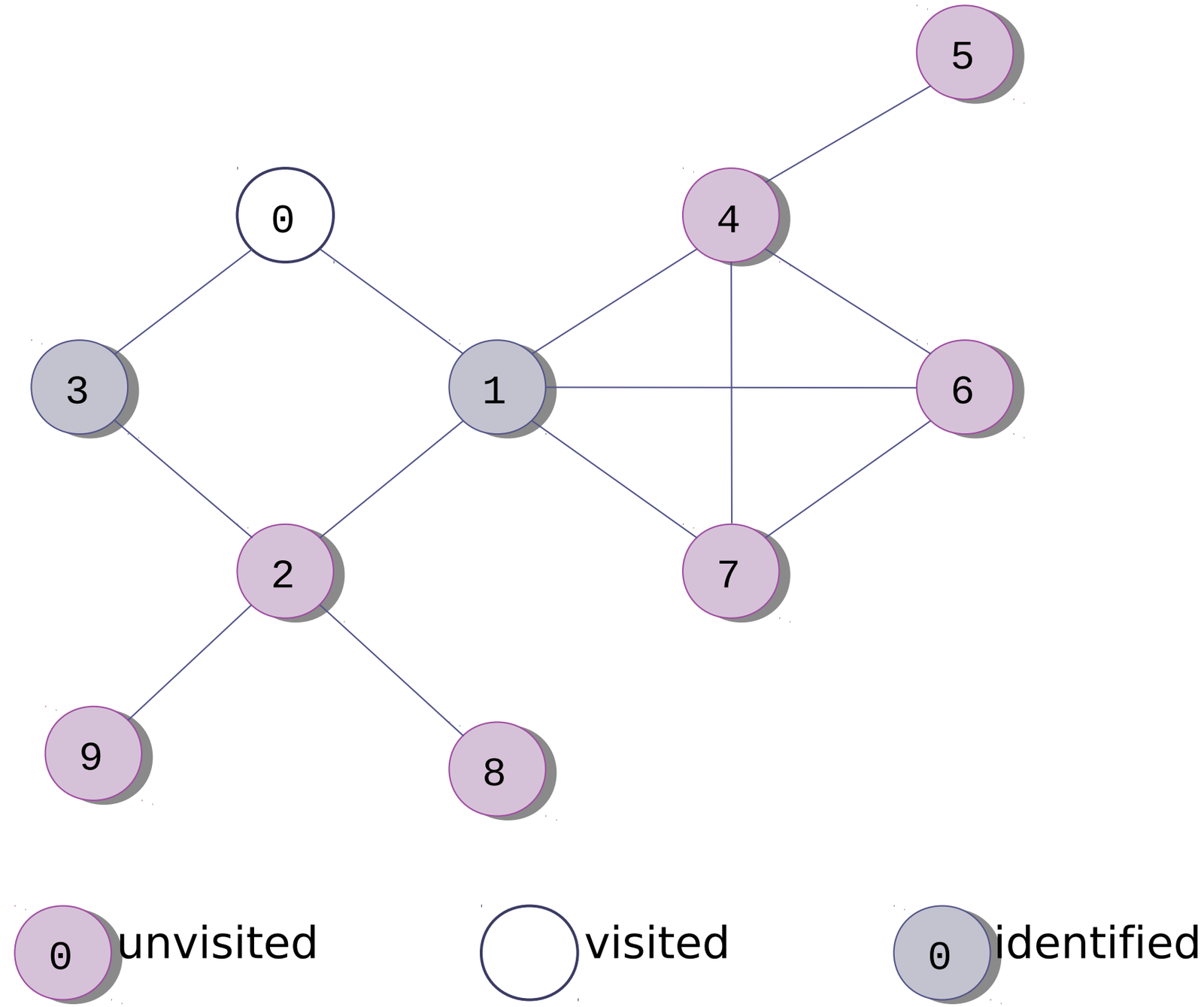


Example of a Breadth-First Search (cont.)

The queue determines which nodes to visit next

Queue:
1, 3

Visit sequence:
0

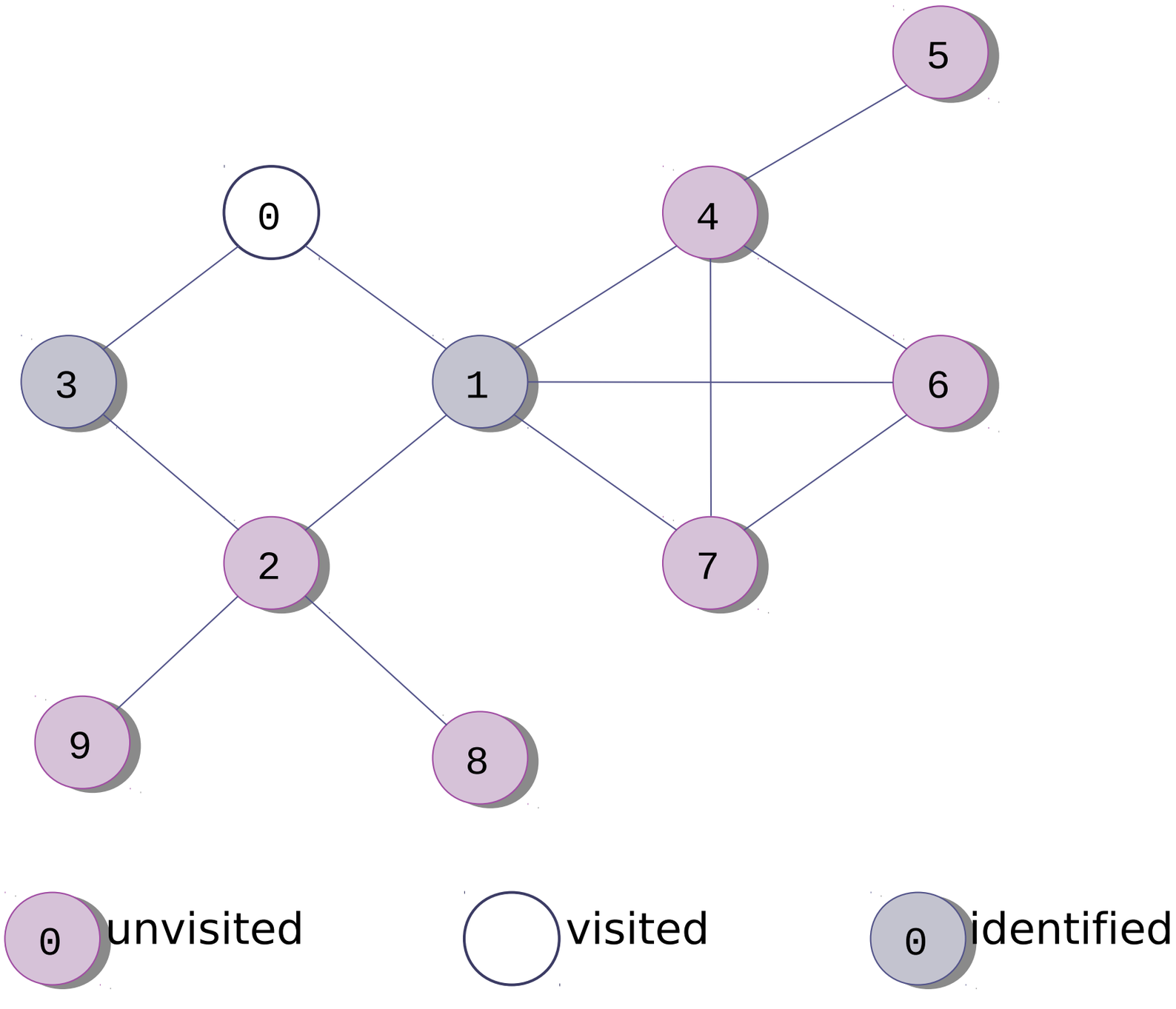


Example of a Breadth-First Search (cont.)

Visit the first node in the queue, 1

Queue:
1, 3

Visit sequence:
0

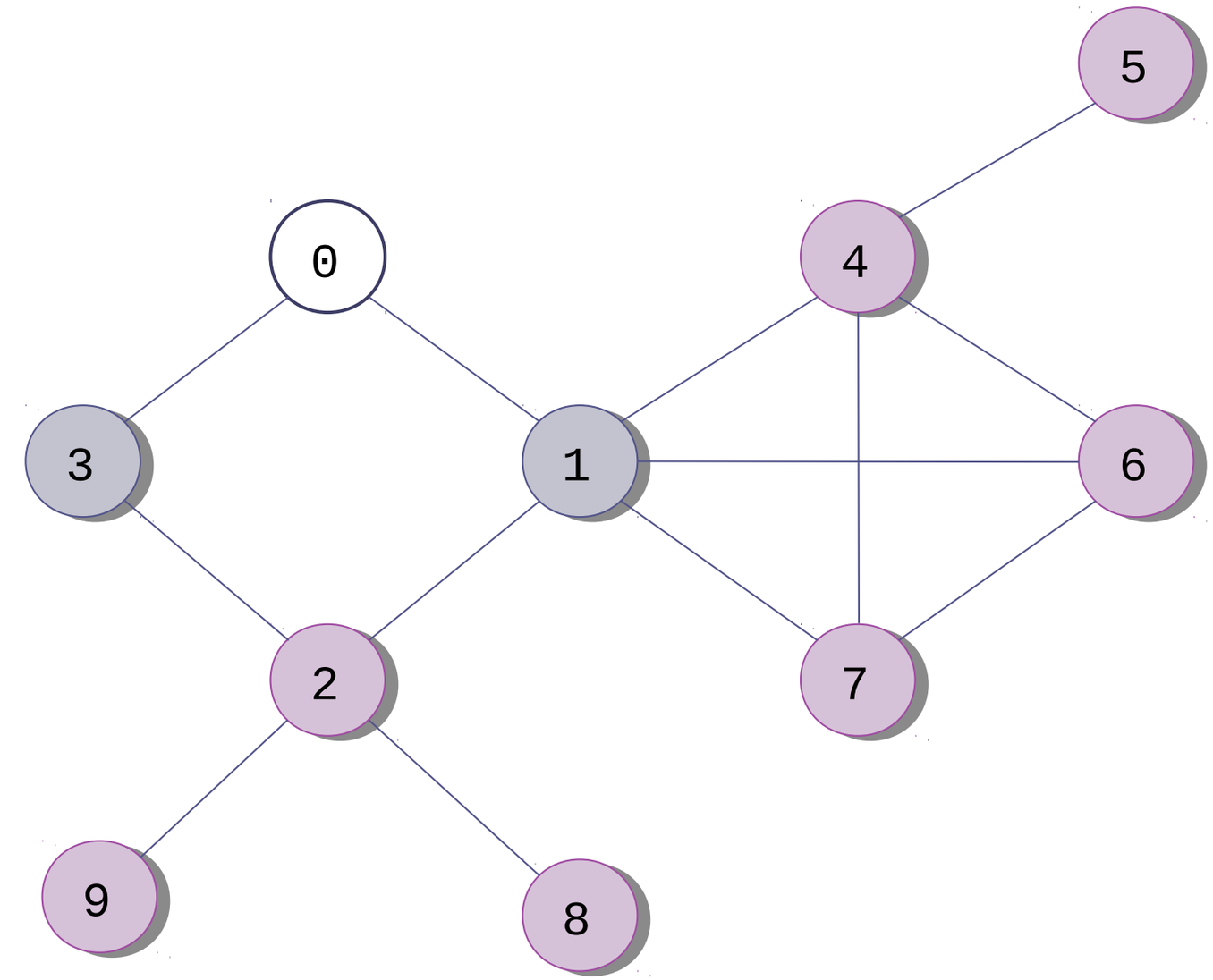


Example of a Breadth-First Search (cont.)

Visit the first node in the queue, 1

Queue:
3

Visit sequence:
0, 1

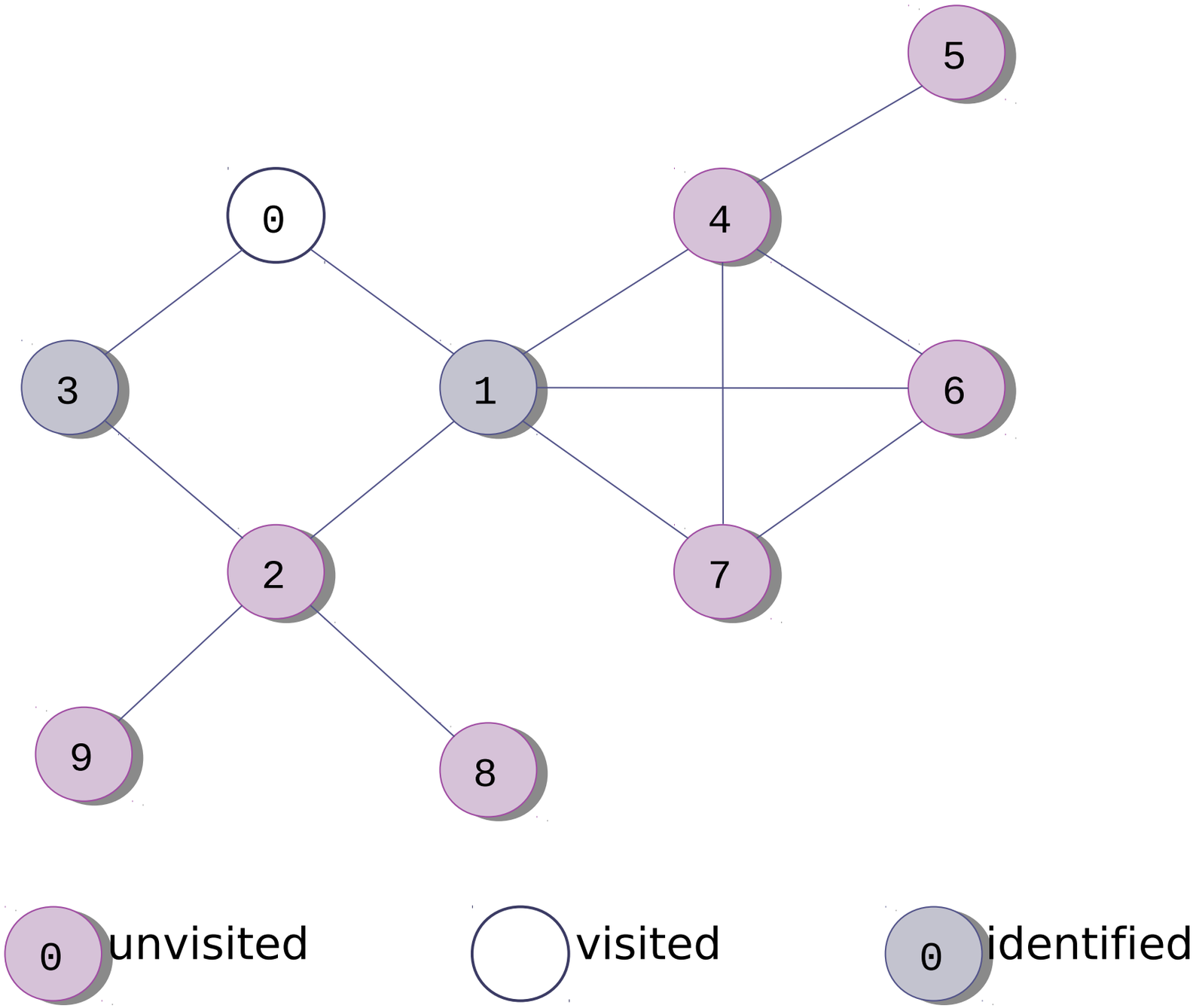


Example of a Breadth-First Search (cont.)

Select all its adjacent nodes that have not been visited or identified

Queue:
3

Visit sequence:
0, 1

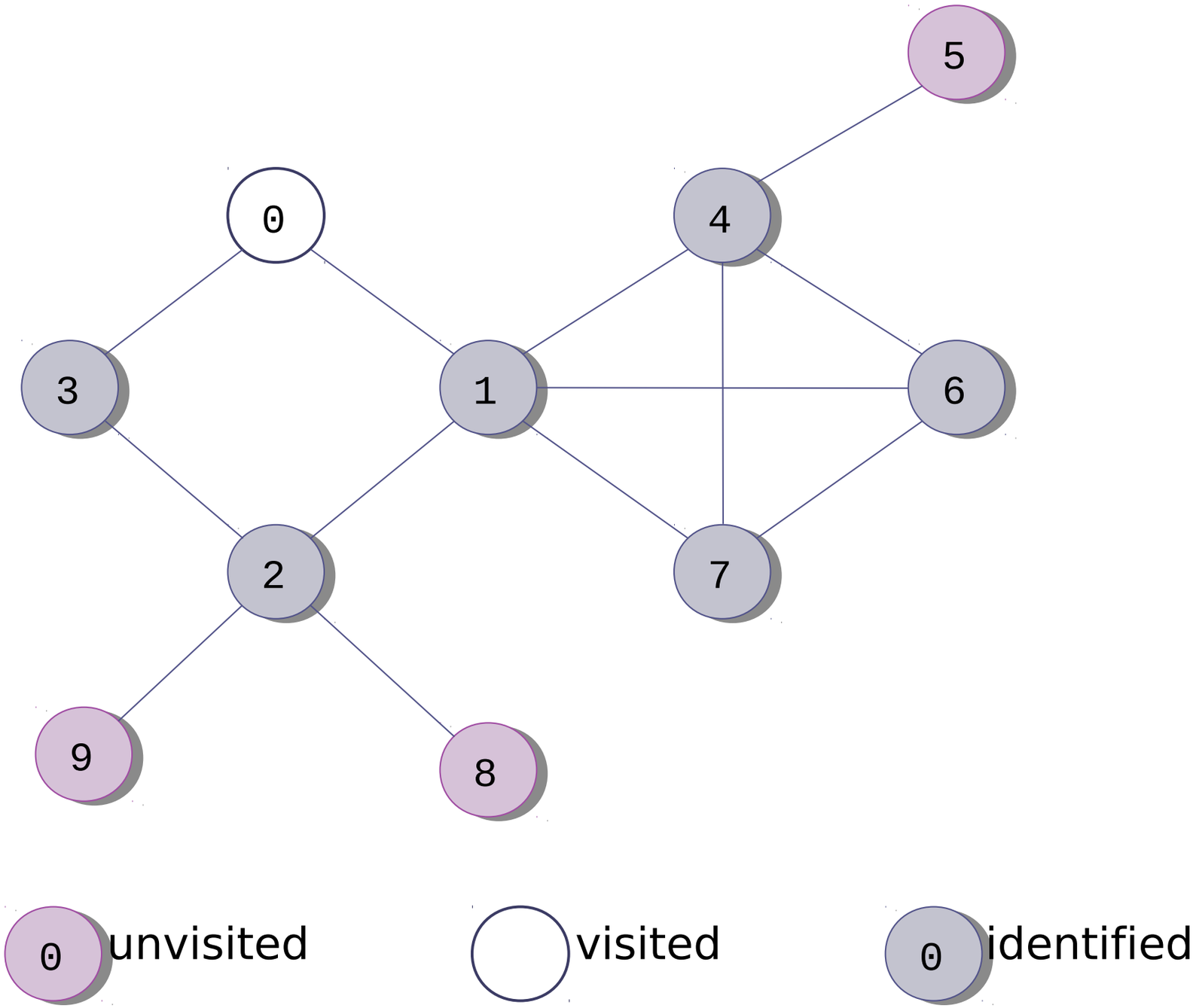


Example of a Breadth-First Search (cont.)

Select all its adjacent nodes that have not been visited or identified

Queue:
3, 2, 4, 6, 7

Visit sequence:
0, 1

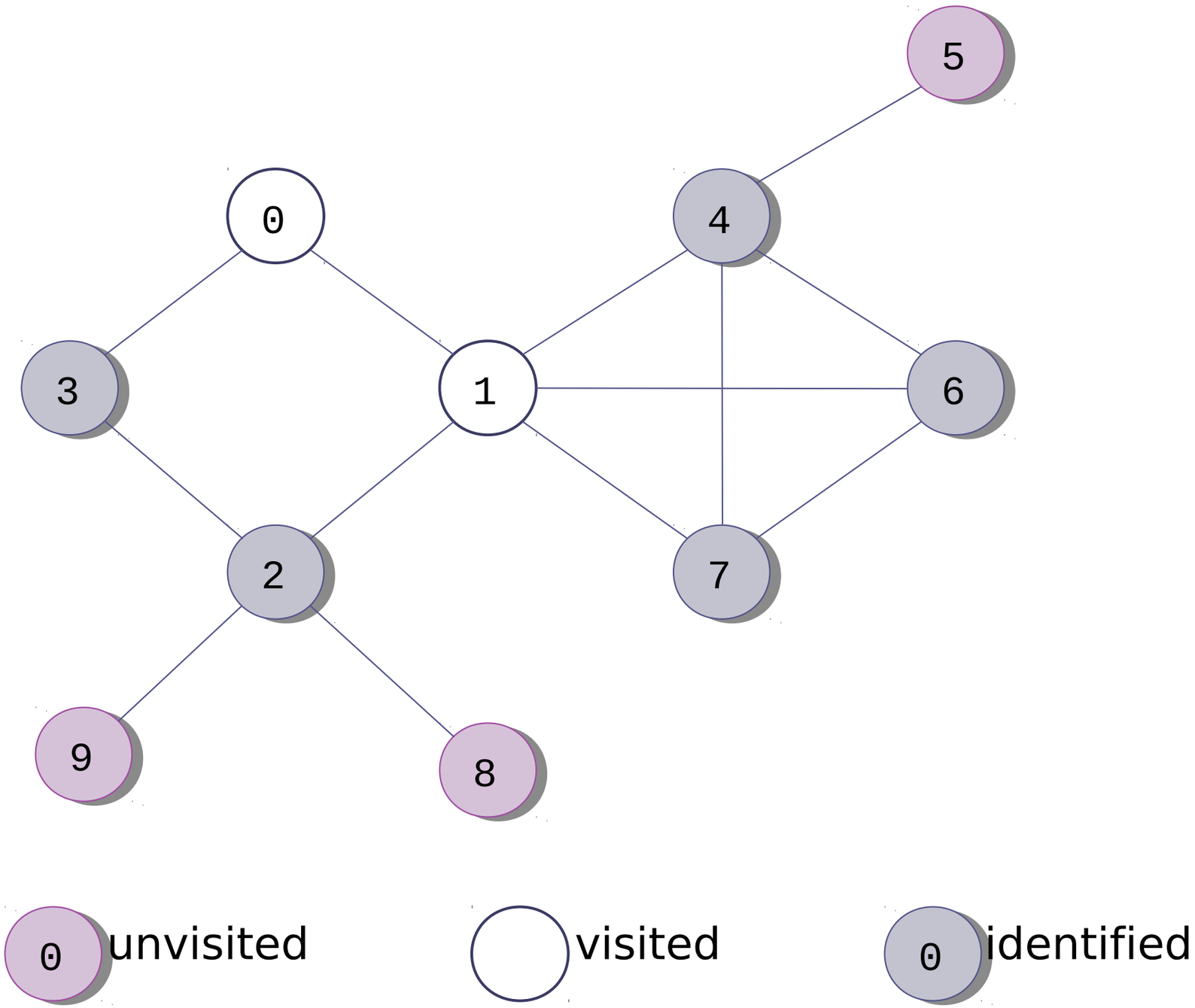


Example of a Breadth-First Search (cont.)

Now that we are done with 1, we color it as visited

Queue:
3, 2, 4, 6, 7

Visit sequence:
0, 1

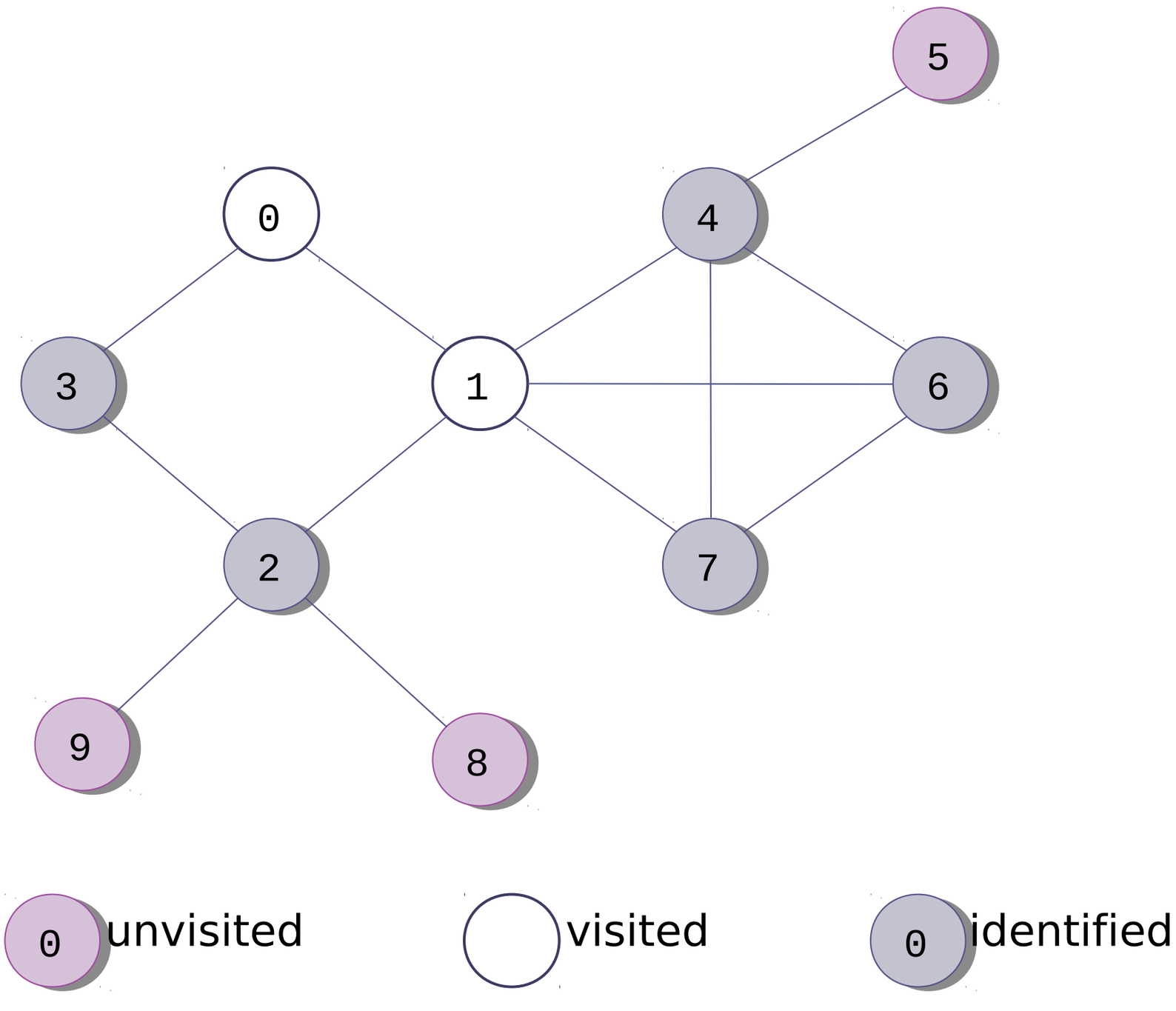


Example of a Breadth-First Search (cont.)

and then visit the next node in the queue, 3 (which was identified in the first selection)

Queue:
3, 2, 4, 6, 7

Visit sequence:
0, 1

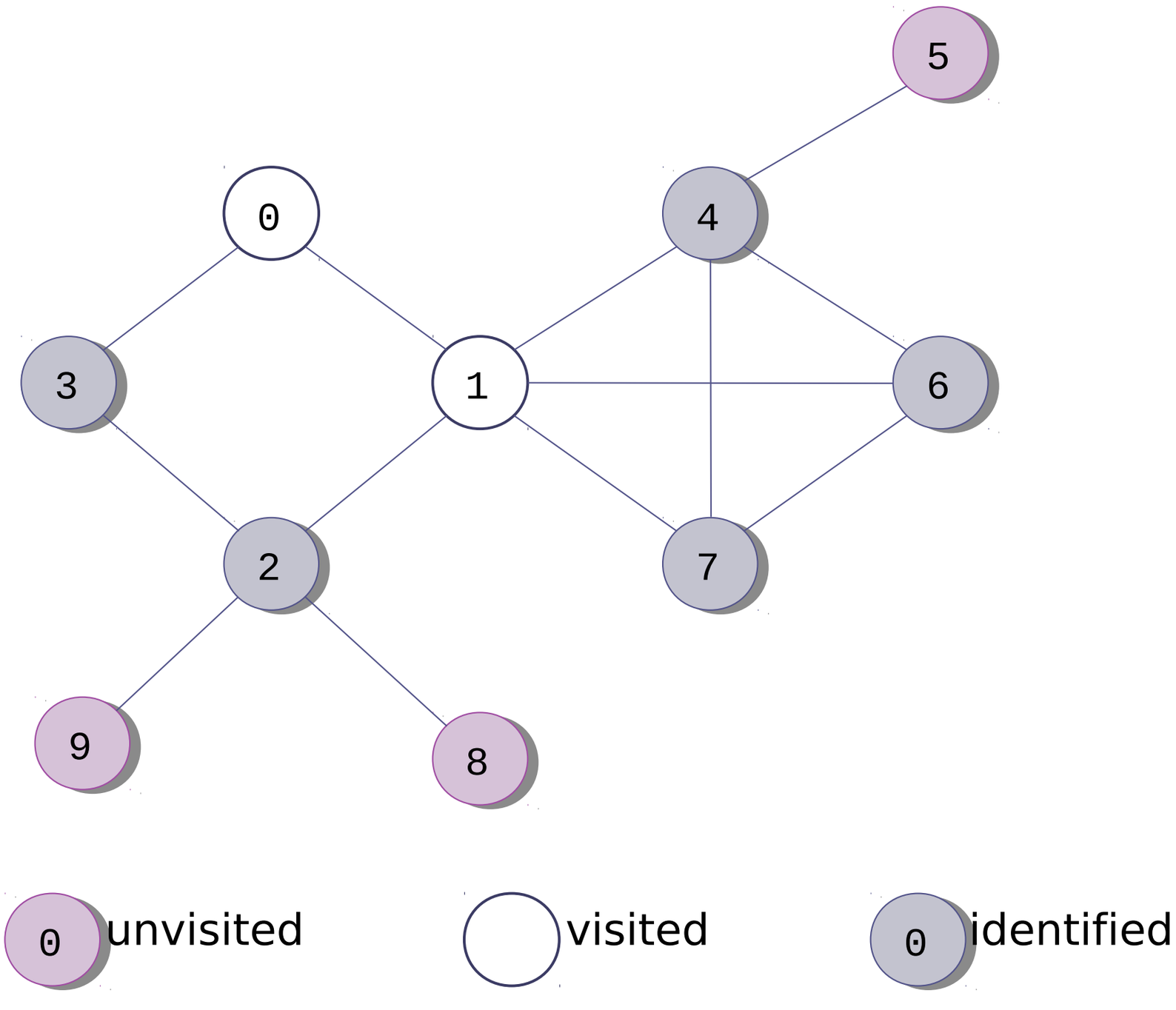


Example of a Breadth-First Search (cont.)

and then visit the next node in the queue, 3 (which was identified in the first selection)

Queue:
2, 4, 6, 7

Visit sequence:
0, 1, 3

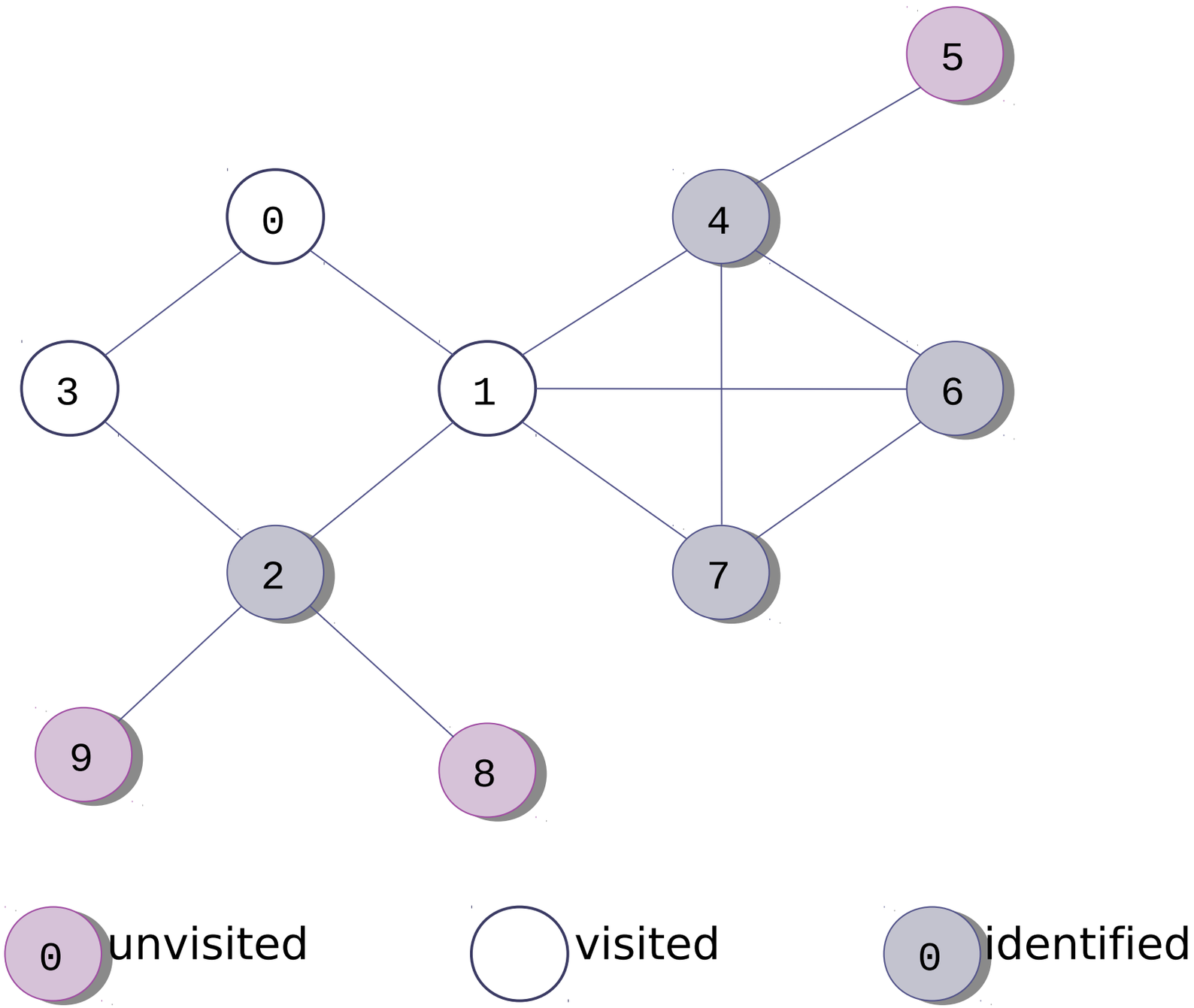


Example of a Breadth-First Search (cont.)

3 has two adjacent vertices. 0 has already been visited and 2 has already been identified. We are done with 3

Queue:
2, 4, 6, 7

Visit sequence:
0, 1, 3

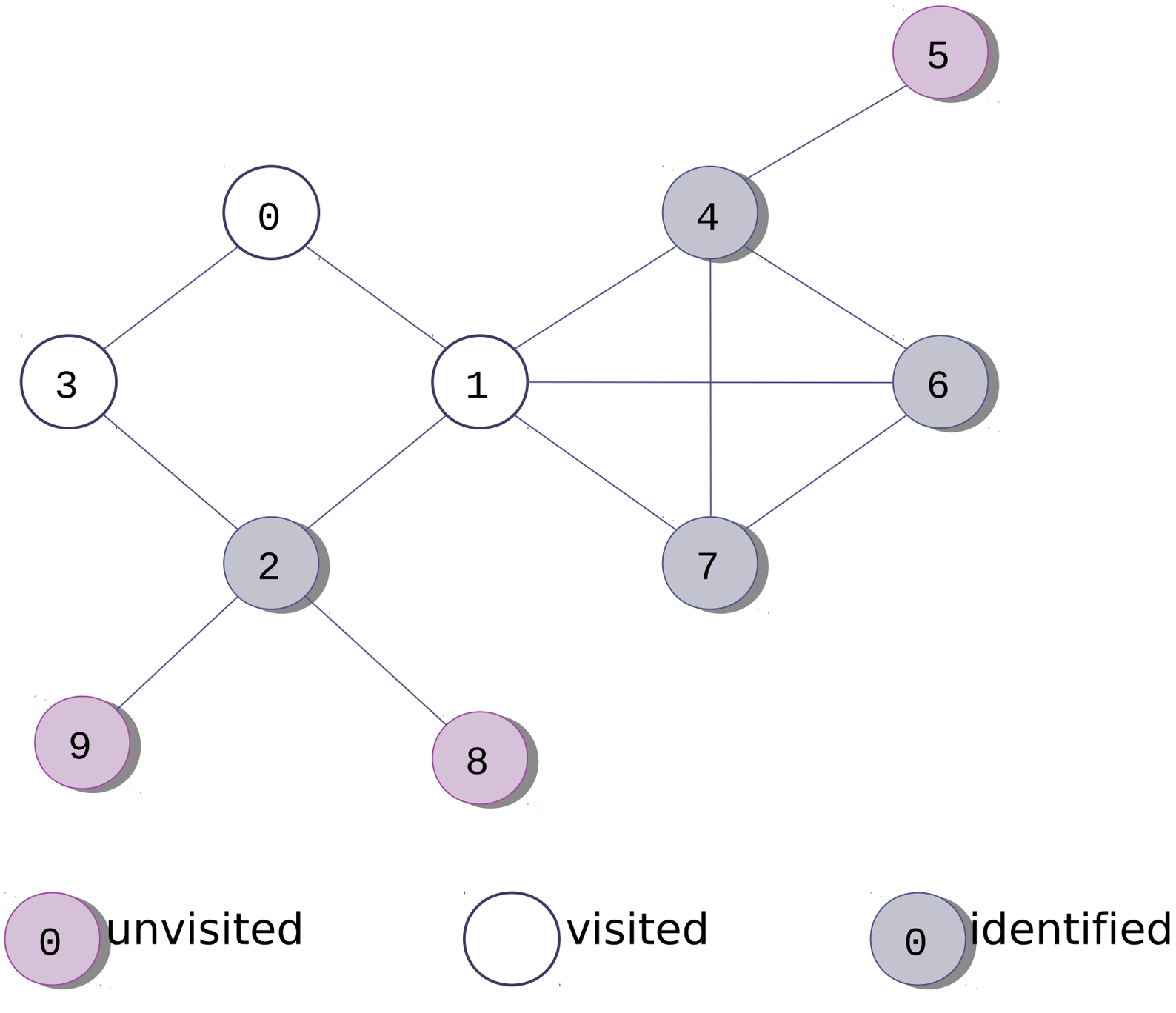


Example of a Breadth-First Search (cont.)

The next node in the queue is 2

Queue:
2, 4, 6, 7

Visit sequence:
0, 1, 3

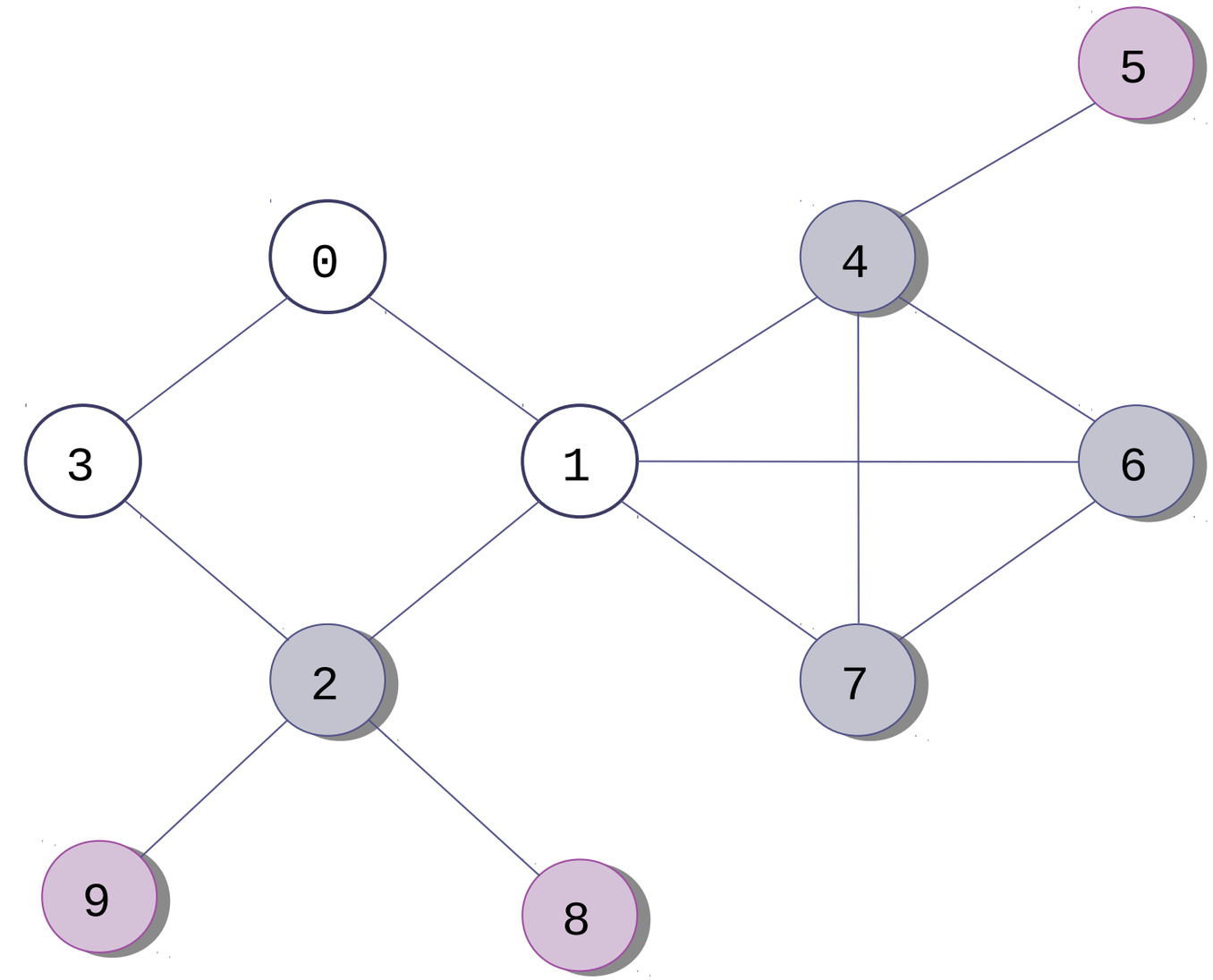


Example of a Breadth-First Search (cont.)

The next node in the queue is 2

Queue:
4, 6, 7

Visit sequence:
0, 1, 3, 2

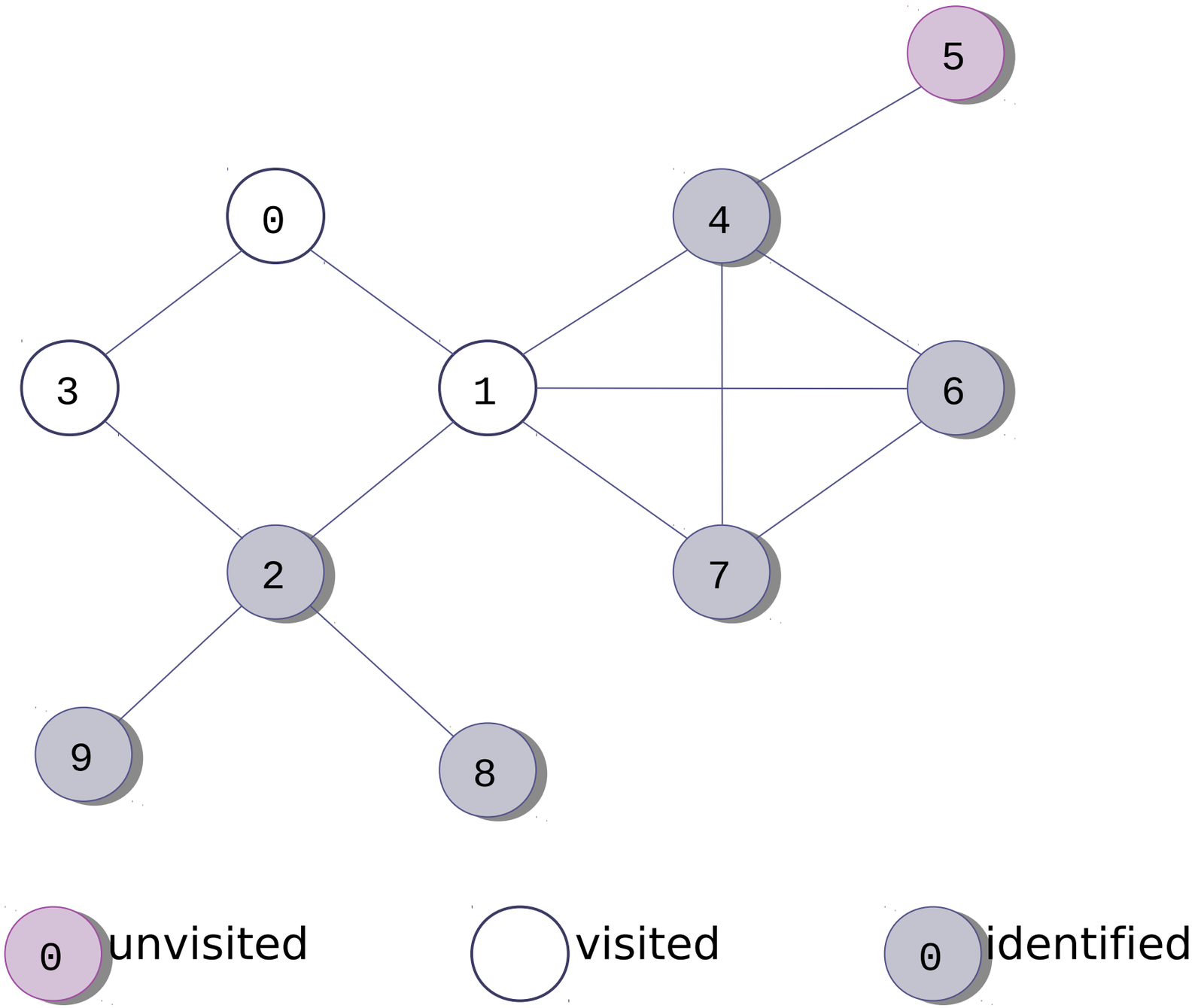


Example of a Breadth-First Search (cont.)

8 and 9 are the only adjacent vertices not already visited or identified

Queue:
4, 6, 7, 8, 9

Visit sequence:
0, 1, 3, 2

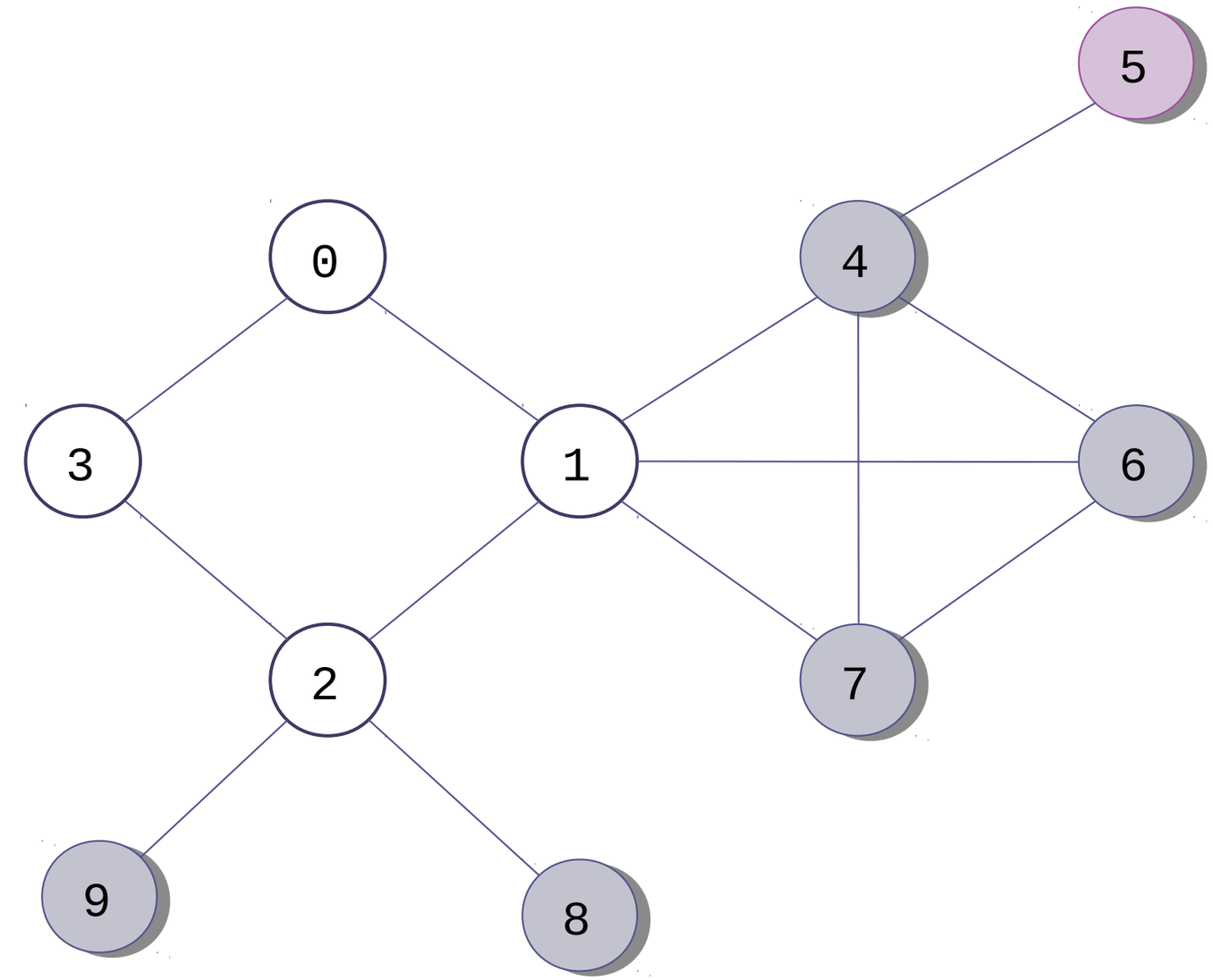


Example of a Breadth-First Search (cont.)

4 is next

Queue:
6, 7, 8, 9

Visit sequence:
0, 1, 3, 2, 4

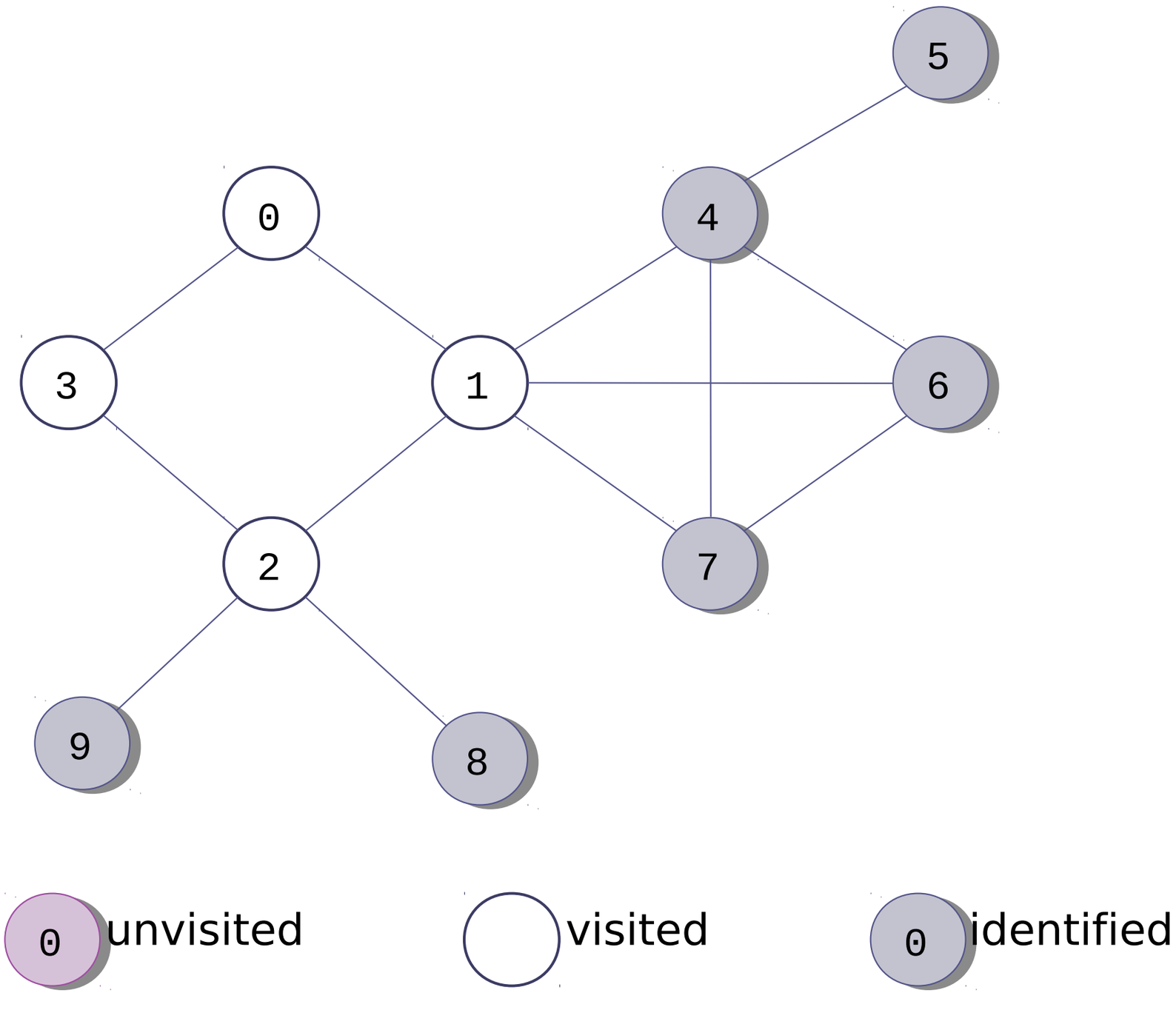


Example of a Breadth-First Search (cont.)

5 is the only vertex not already visited or identified

Queue:
6, 7, 8, 9, 5

Visit sequence:
0, 1, 3, 2, 4

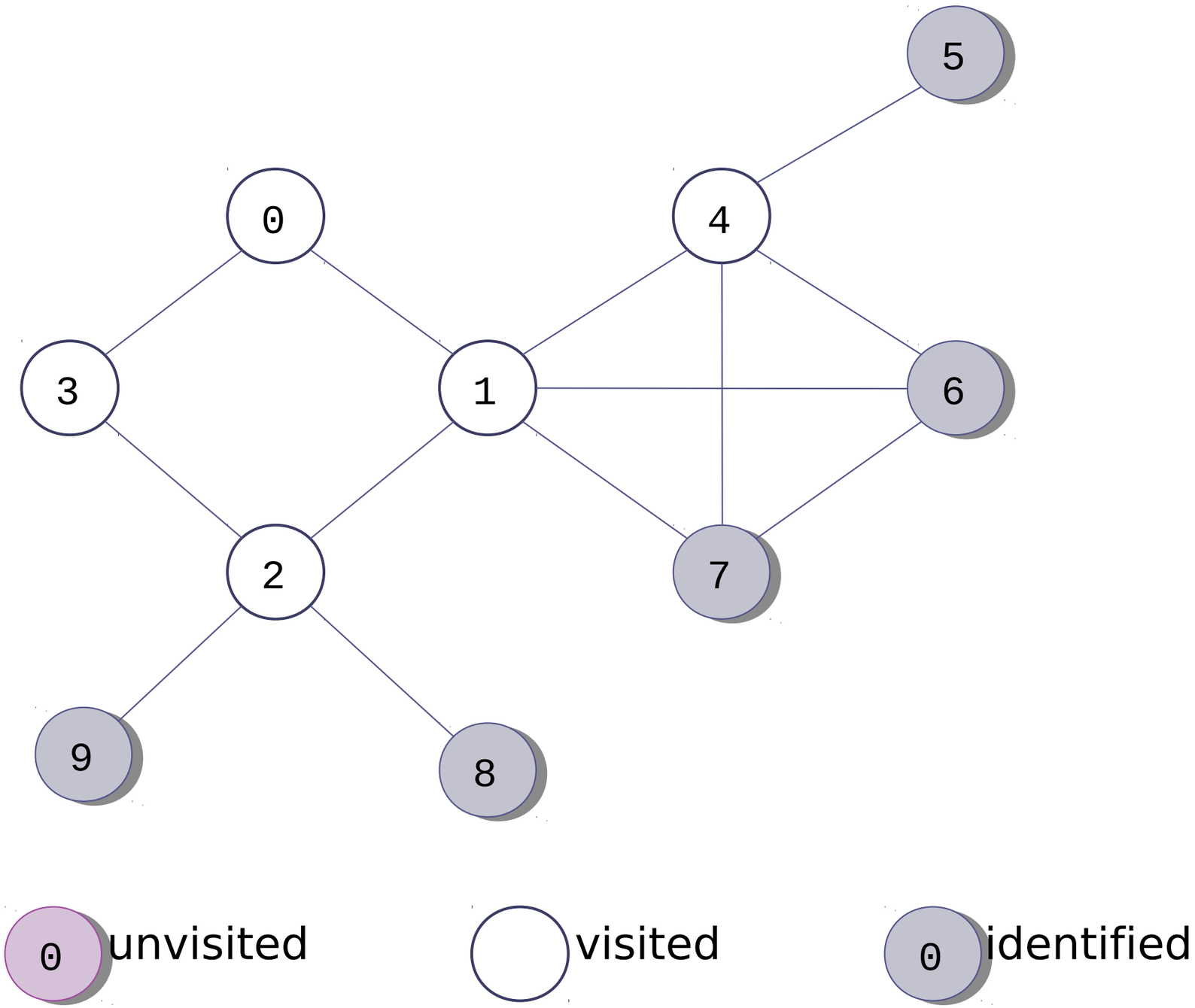


Example of a Breadth-First Search (cont.)

6 has no vertices not already visited or identified

Queue:
7, 8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6

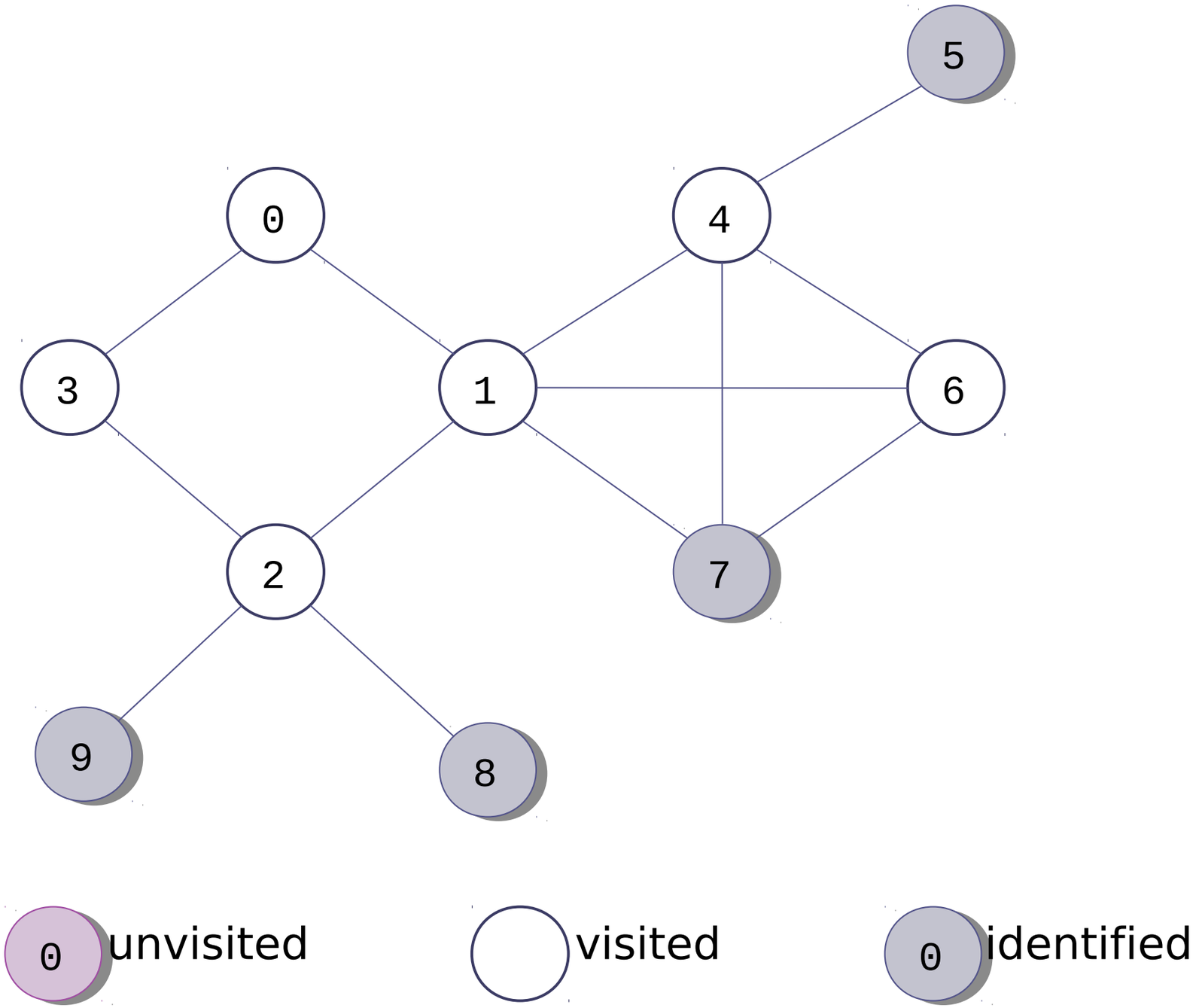


Example of a Breadth-First Search (cont.)

6 has no vertices not already visited or identified

Queue:
7, 8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6

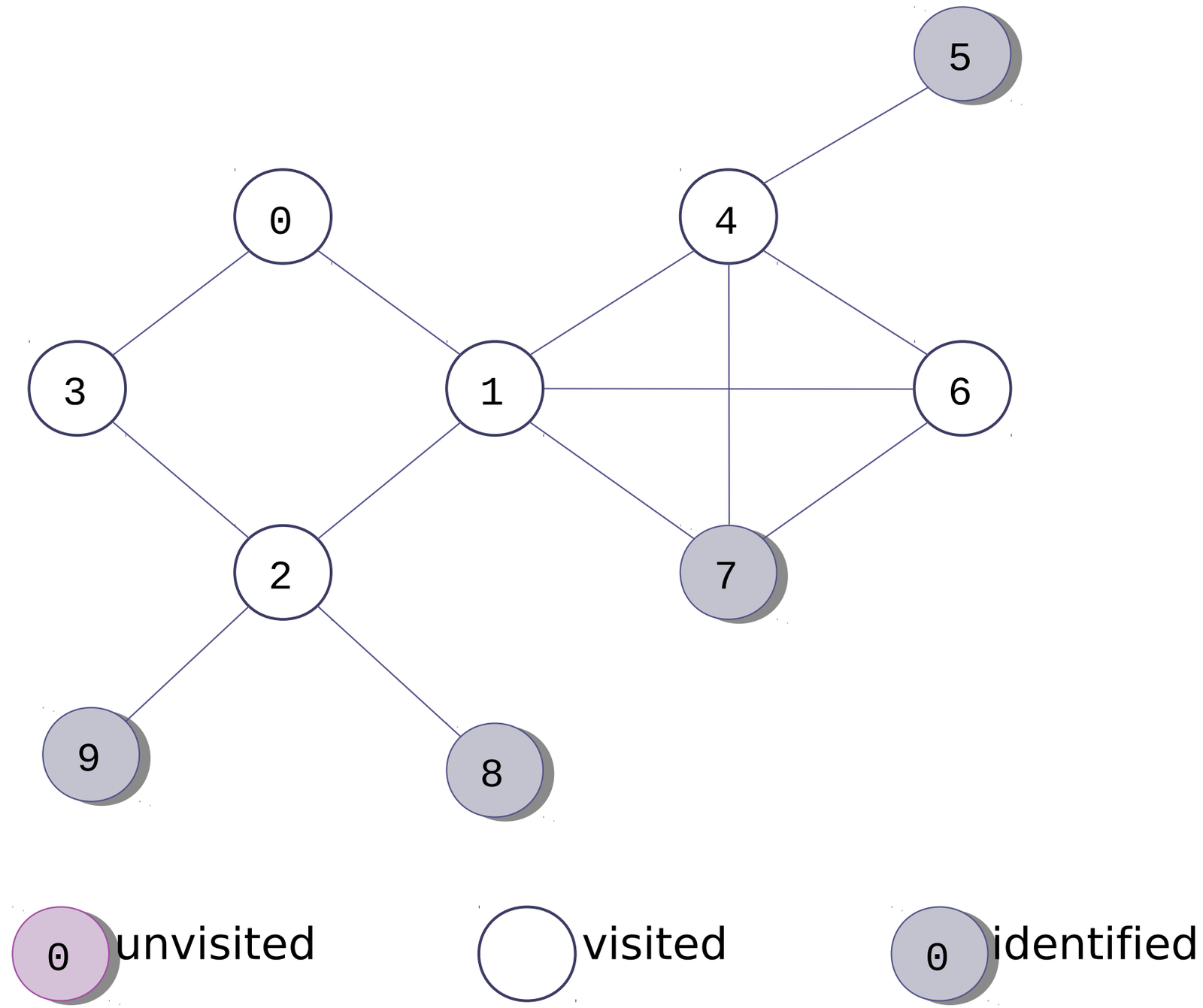


Example of a Breadth-First Search (cont.)

7 has no vertices not already visited or identified

Queue:
8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6, 7

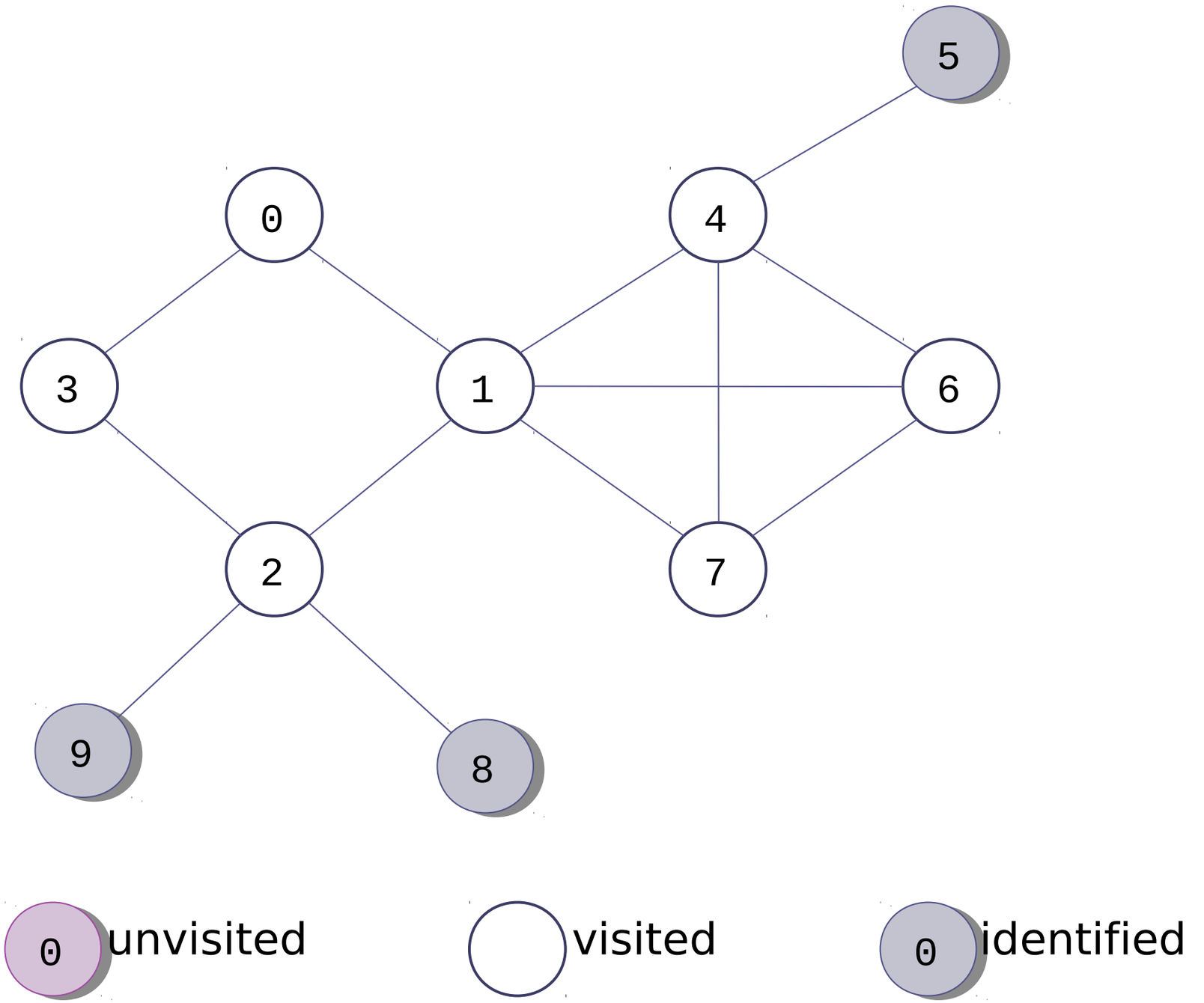


Example of a Breadth-First Search (cont.)

7 has no vertices not already visited or identified

Queue:
8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6, 7

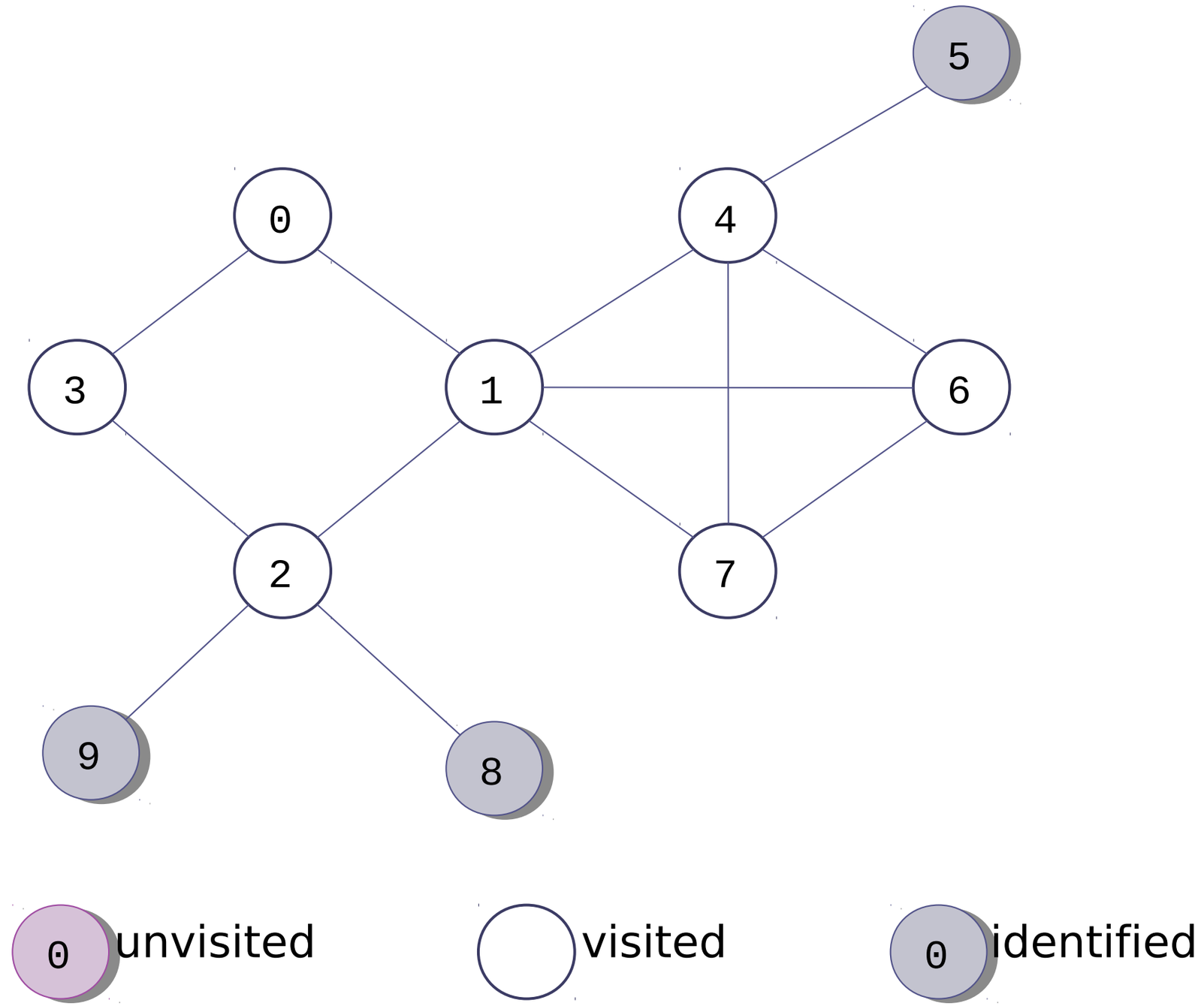


Example of a Breadth-First Search (cont.)

We go back to the vertices of 2 and visit them

Queue:
8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6, 7

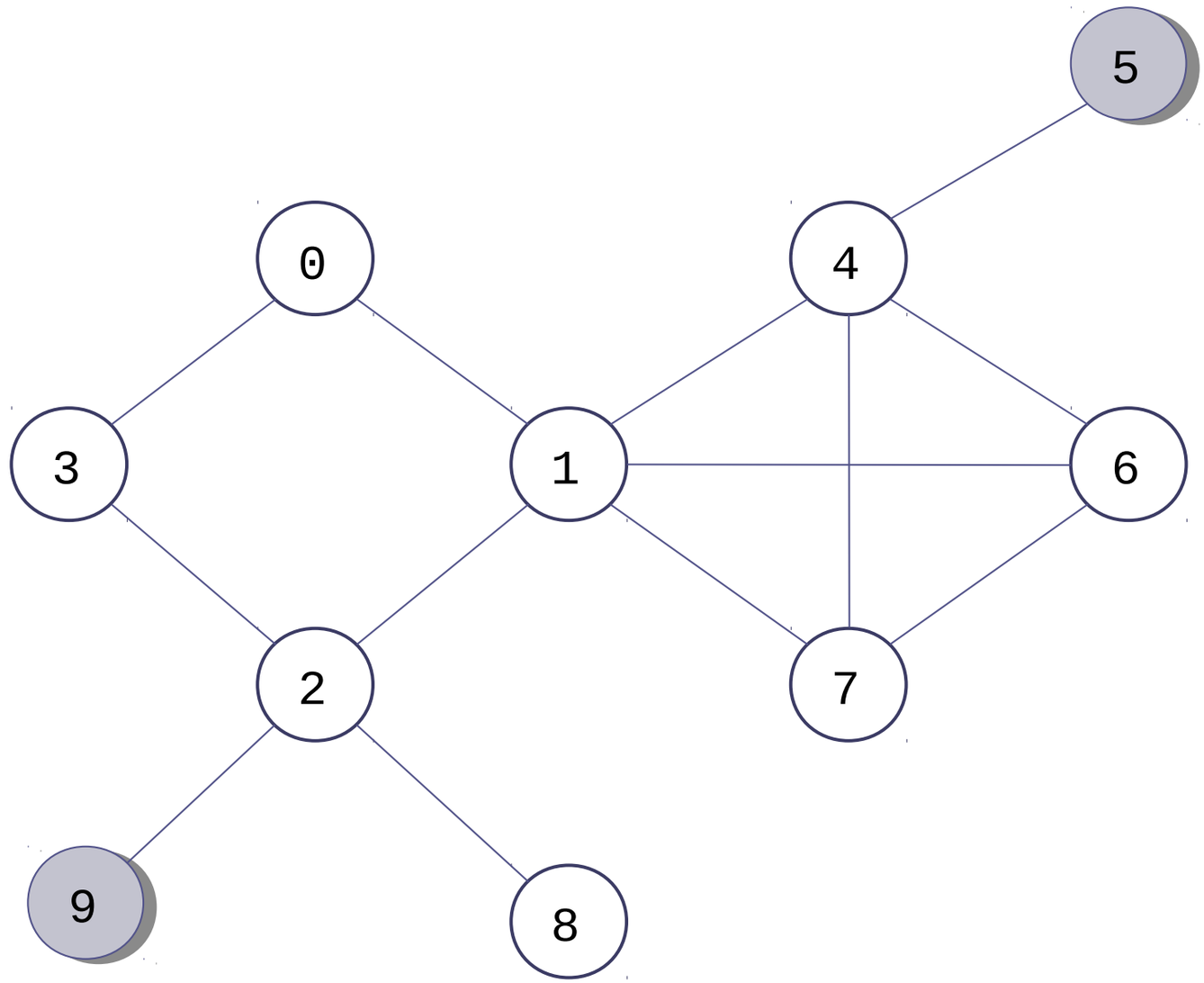


Example of a Breadth-First Search (cont.)

8 has no vertices not already visited or identified

Queue:
9, 5

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8

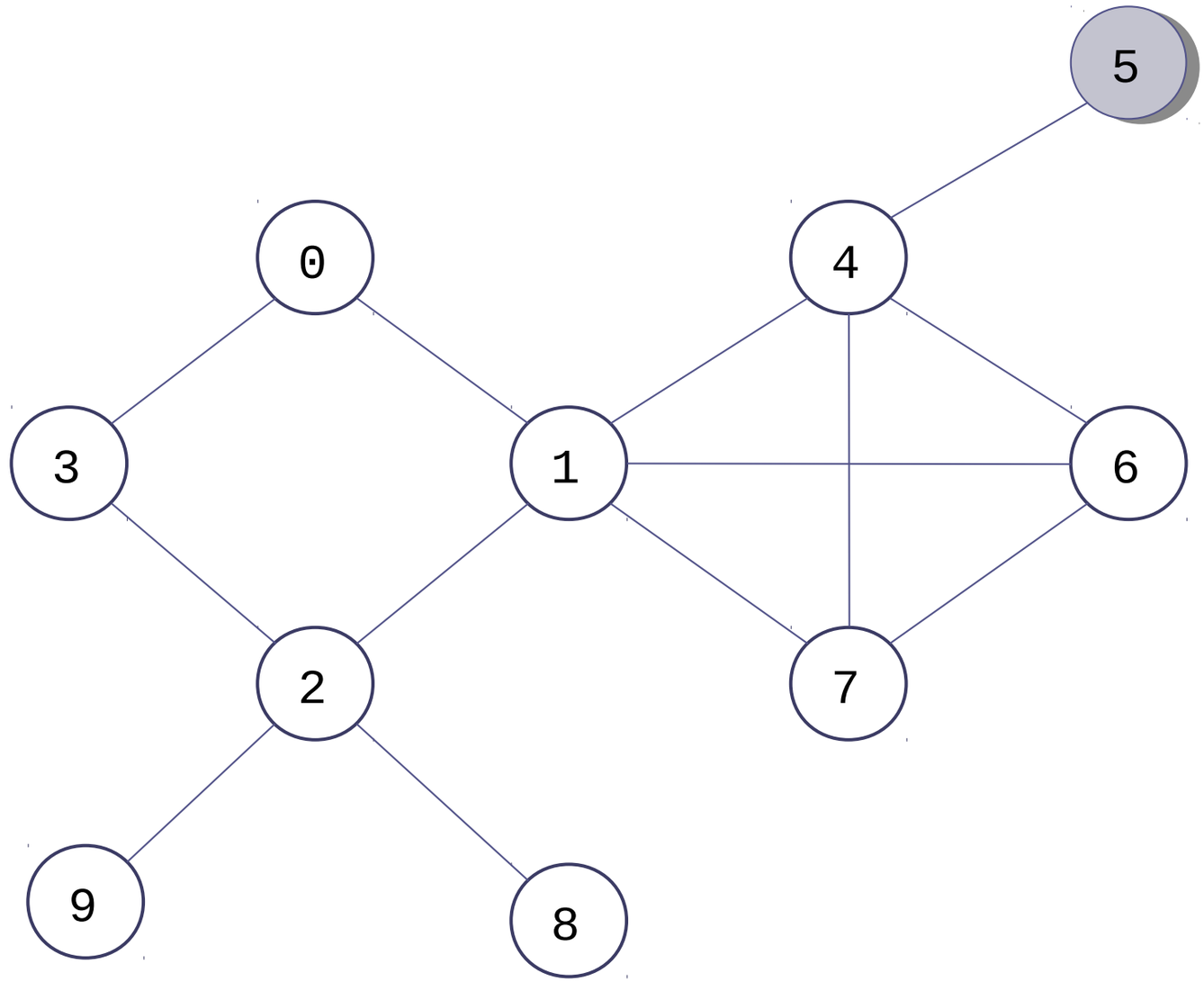


Example of a Breadth-First Search (cont.)

9 has no vertices not already visited or identified

Queue:
5

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9

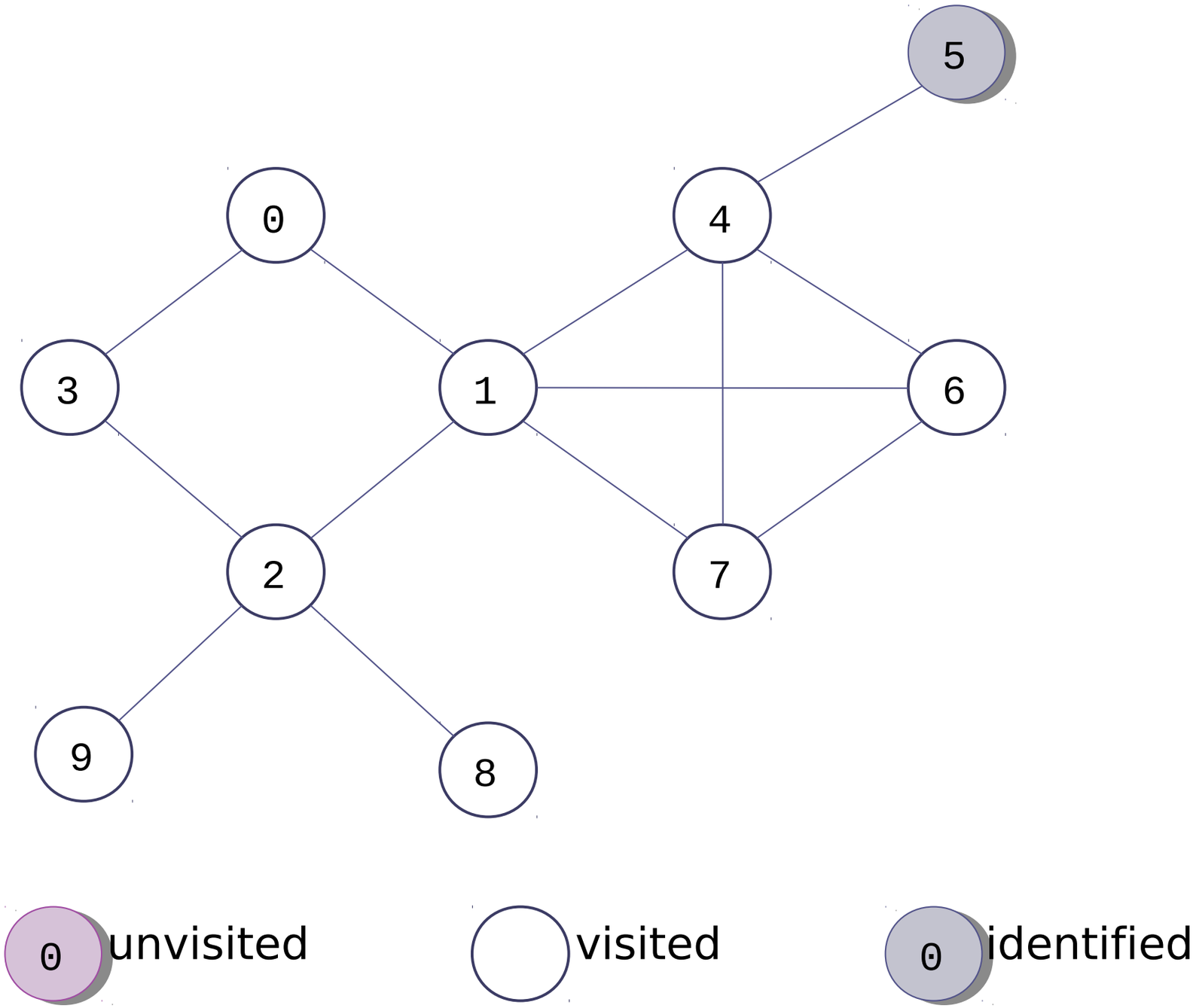


Example of a Breadth-First Search (cont.)

Finally we visit 5

Queue:
5

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9

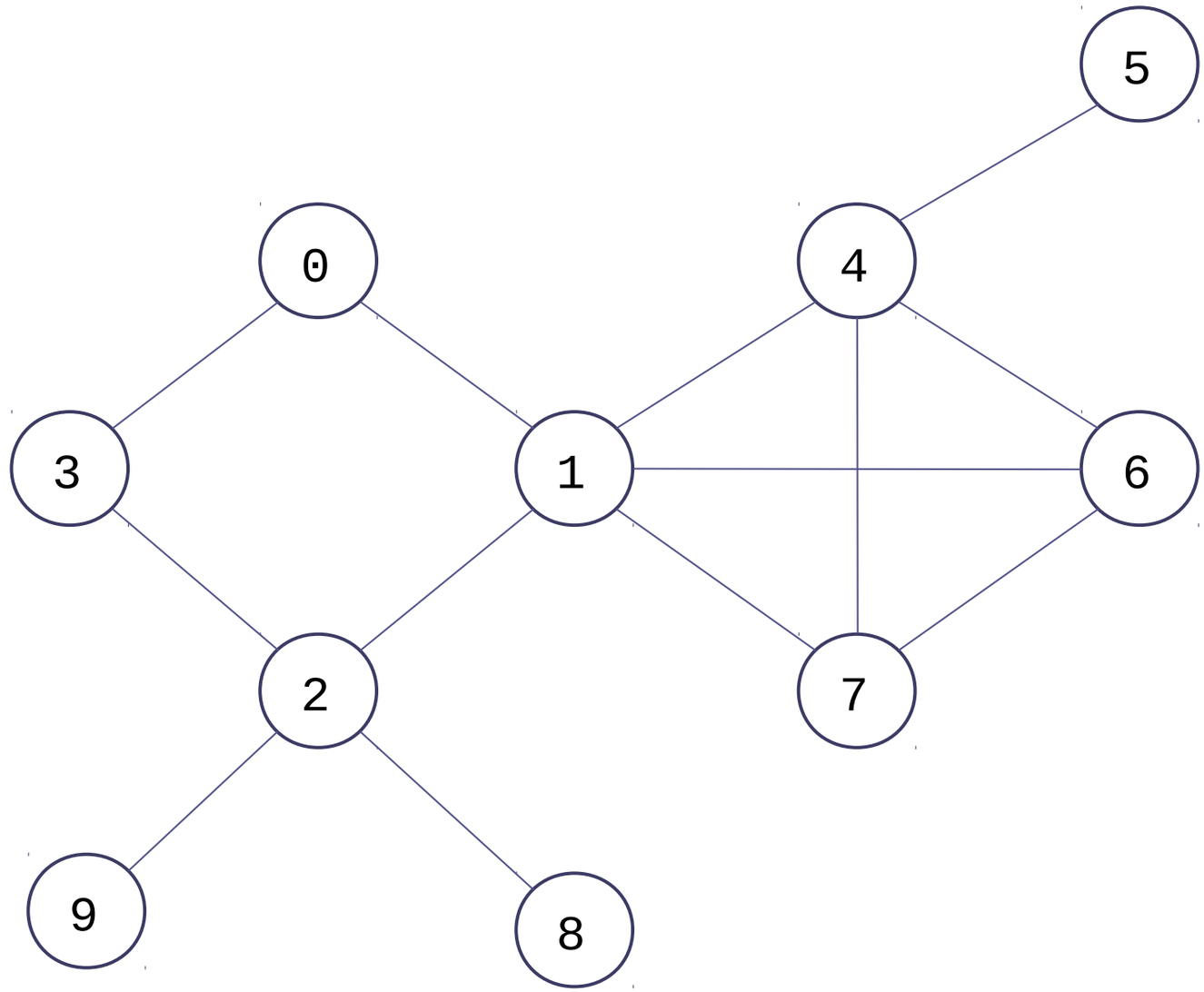


Example of a Breadth-First Search (cont.)

which has no vertices not already visited or identified

Queue:
empty

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9, 5

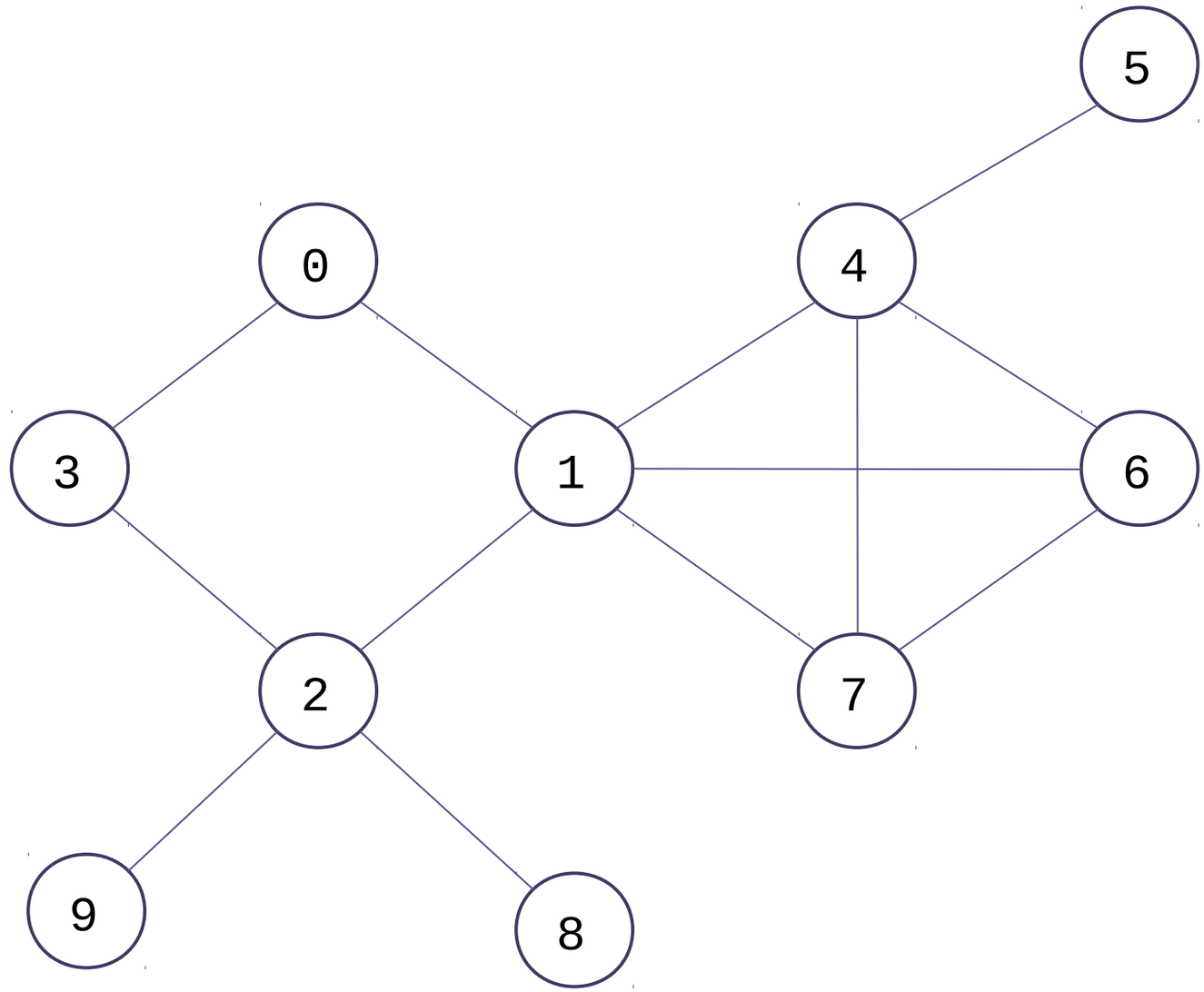


Example of a Breadth-First Search (cont.)

The queue is empty; all vertices have been visited

Queue:
empty

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9, 5



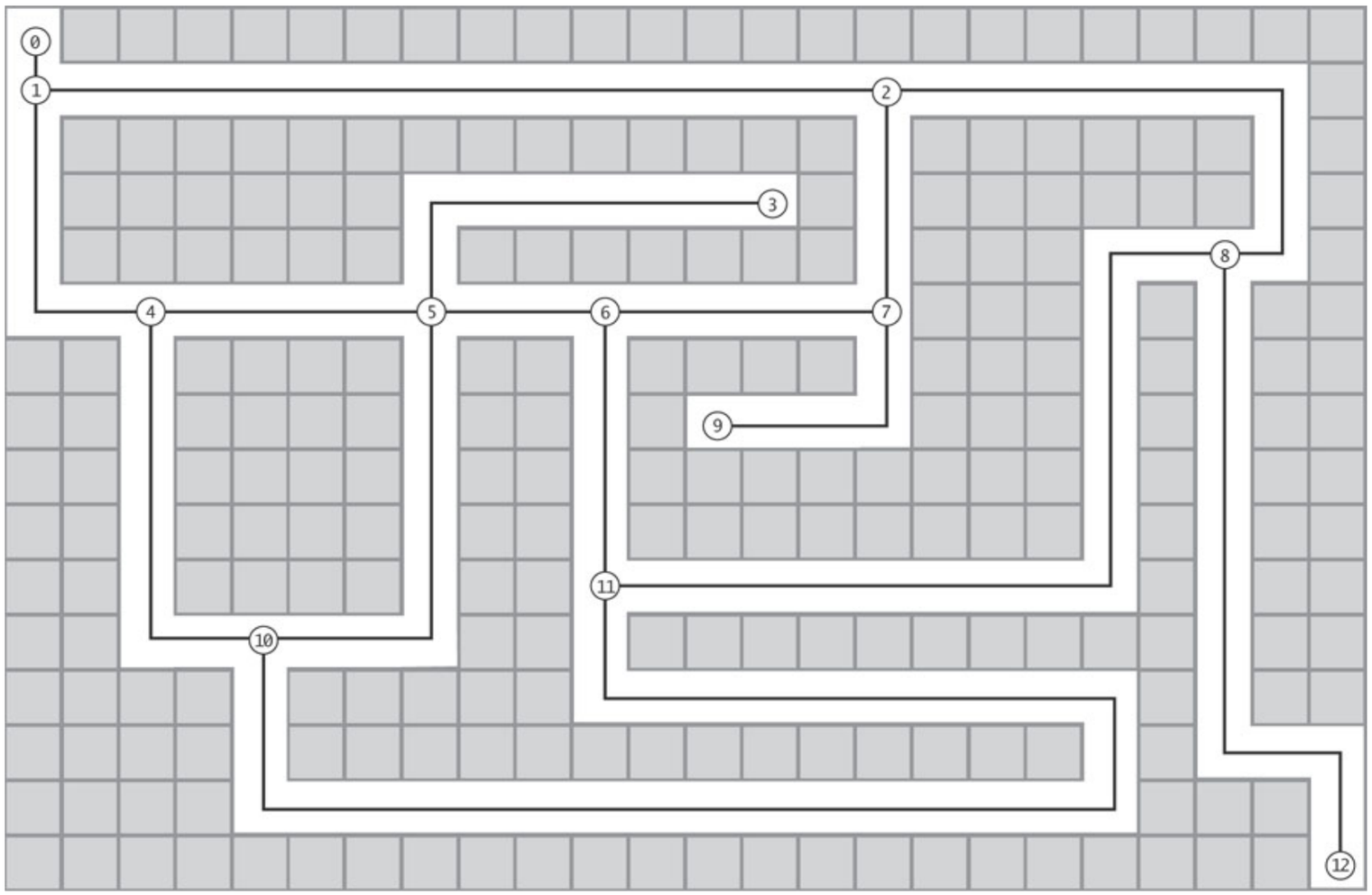
Algoritm för bredden-först

Algorithm for Breadth-First Search

1. Take an arbitrary start vertex, mark it identified (color it light blue), and place it in a queue.
2. `while` the queue is not empty
3. Take a vertex, u, out of the queue and visit u.
4. `for` all vertices, v, adjacent to this vertex, u
5. `if` v has not been identified or visited
6. Mark it identified (color it light blue).
7. Insert vertex v into the queue.
8. We are now finished visiting u (color it dark blue).

Exempel: Kortaste vägen

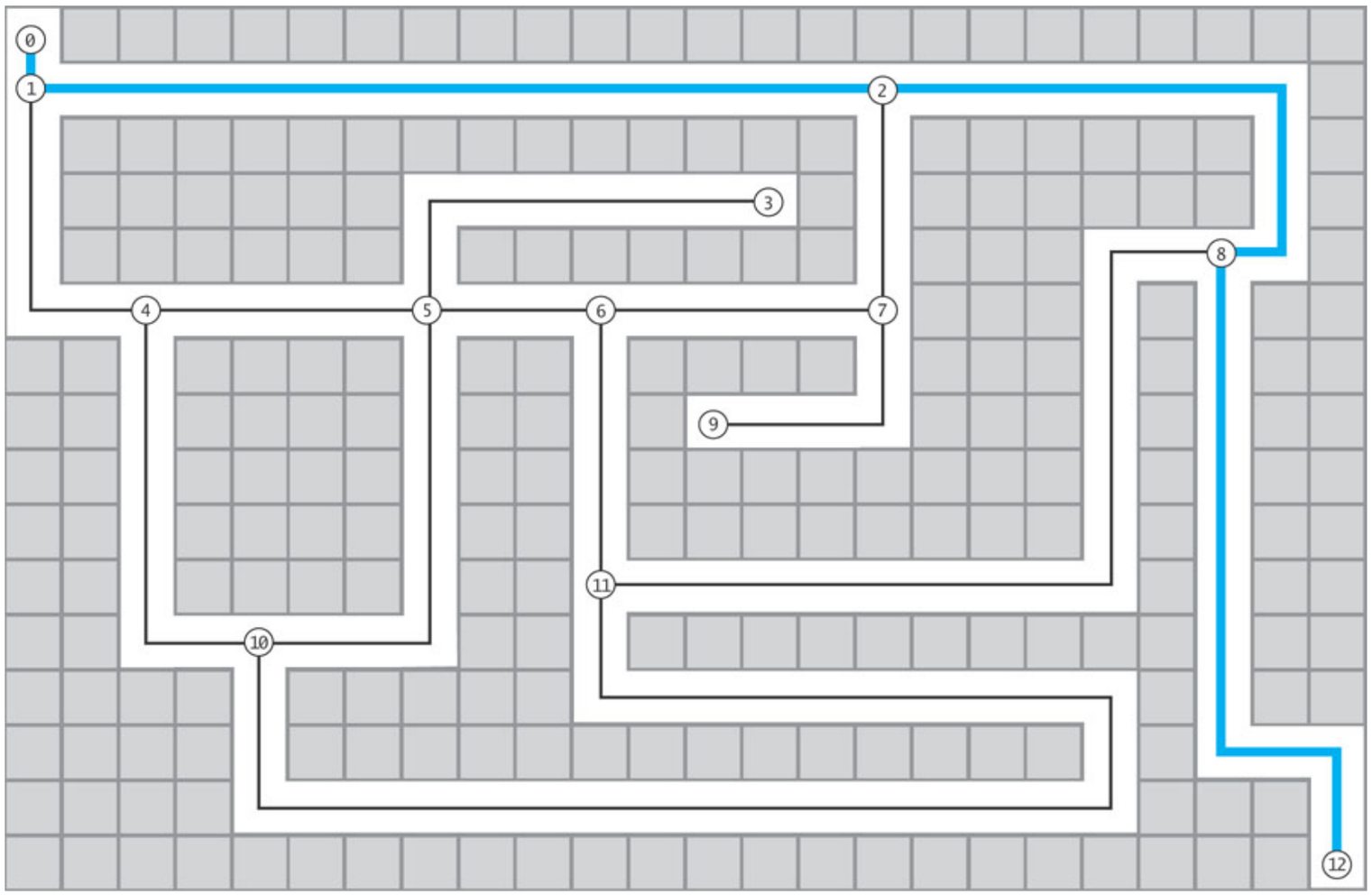
För att hitta kortaste vägen genom en labyrint, kan vi representera labyrinten som en graf:



Exempel: Kortaste vägen

Bredden-först-sökning ger den kortaste vägen:

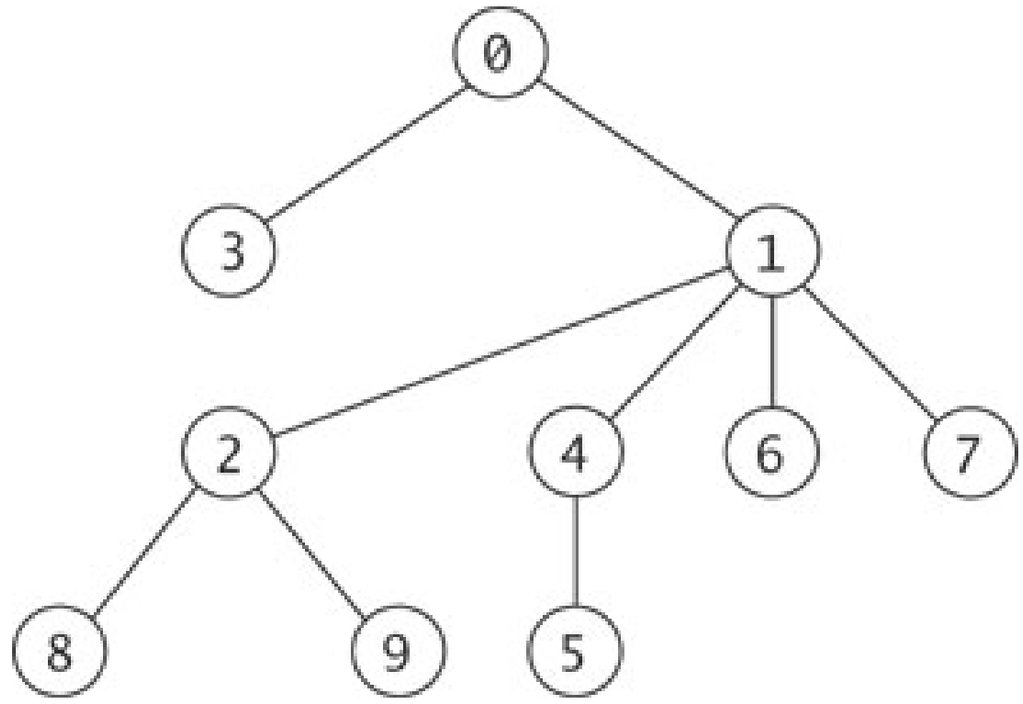
- dvs, minsta antalet korsningar, inte celler!



Bredden-först-sökträdet

Vi kan bygga ett träd som består av de bågar som vi faktiskt utnyttjade vid sökningen.

Detta träd har alla noder och en del av bågarna från originalgrafen.



Informationen som behövs för att representera sökträdet, kan lagras i ett fält: Där lagrar vi föräldern till varje båge när den identifieras.

Vi kan förfina steg 7 i algoritmen såhär:

- 7.1. Stoppa in nod *v* i kön
- 7.2. Sätt *v*:s förälder till *u*

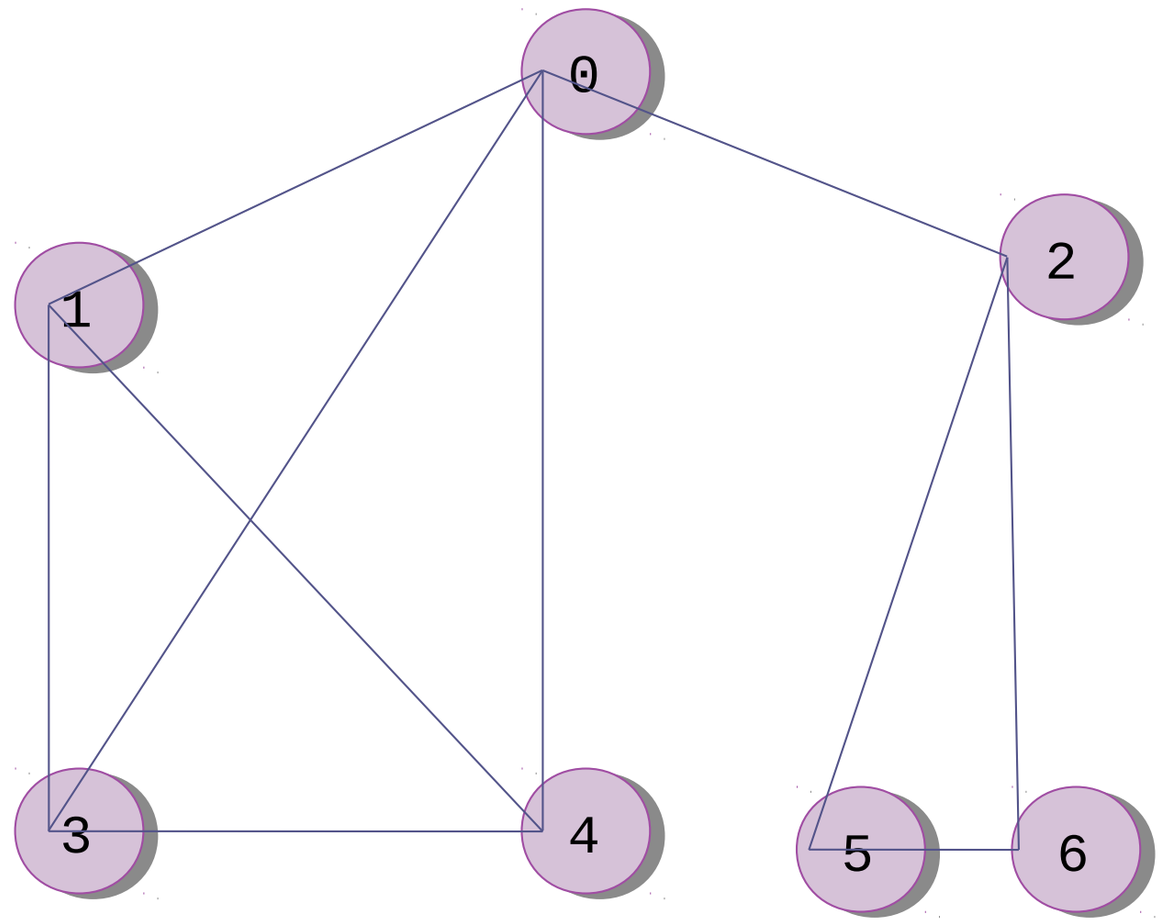
DFS: Djupet-först-sökning

Vid djupet-först-sökning så besöker vi noderna i följande ordning:

- besök startnoden först
- välj en angränsande nod att besöka
- sedan en angränsande nod till denna
- och så vidare tills det inte finns några fler noder
- sedan backar vi och kollar ifall vi kan hitta en annan angränsande båge

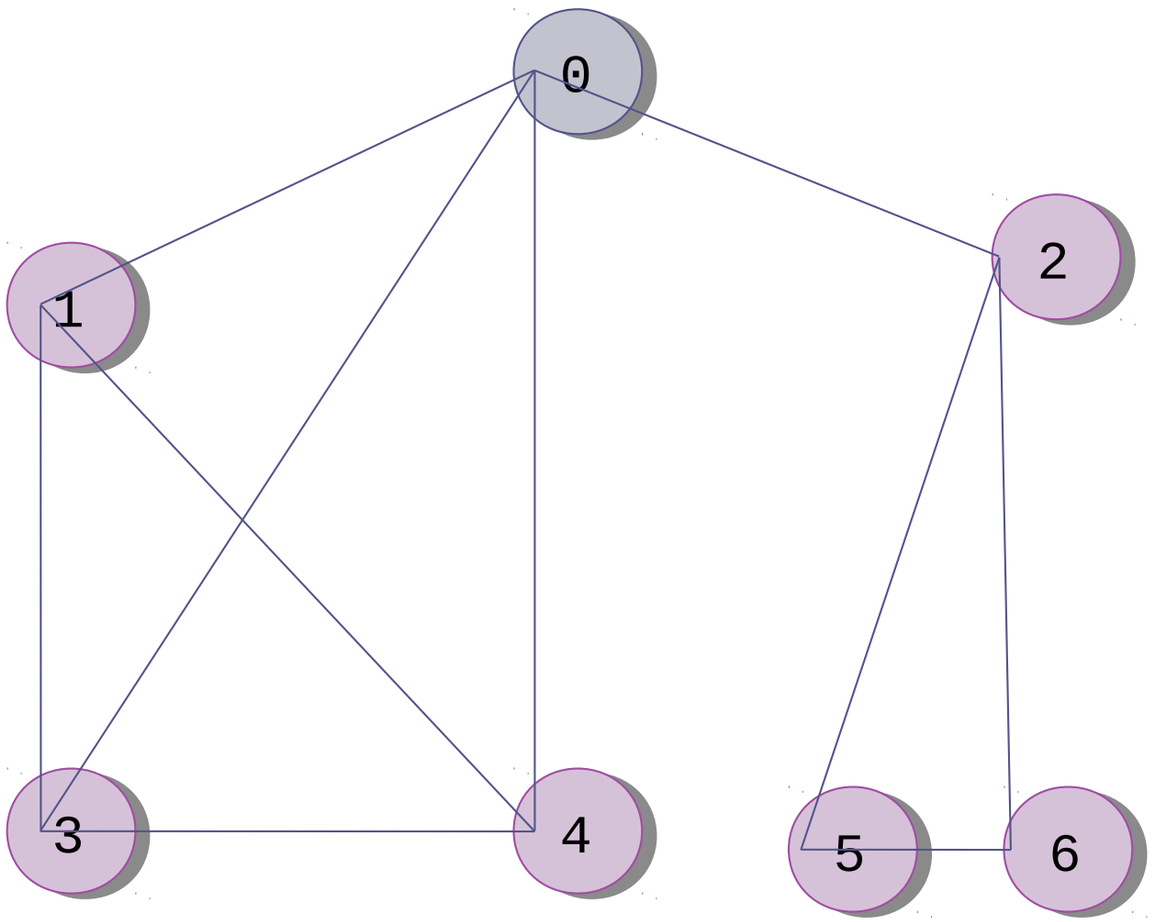
- och så vidare

Example of a Depth-First Search



Example of a Depth-First Search (cont.)

Mark 0 as being visited

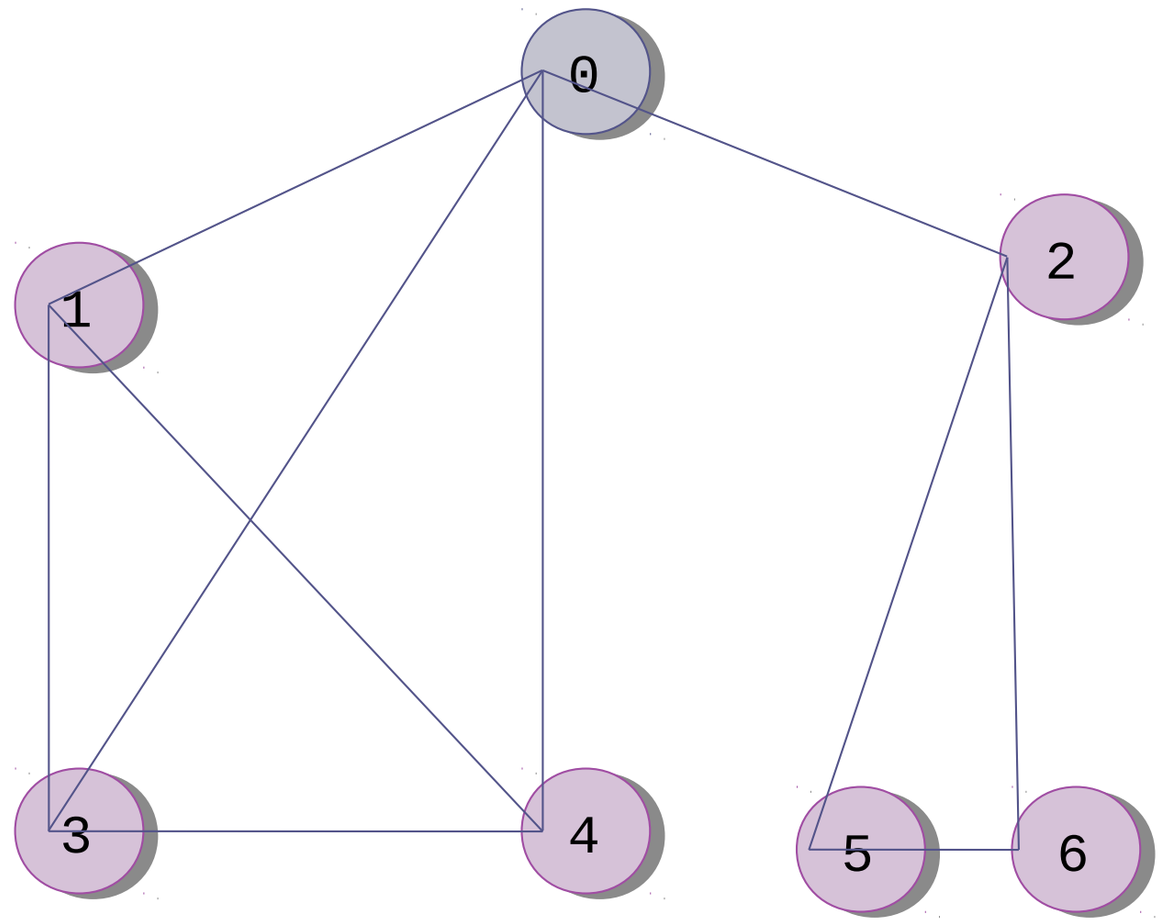


Discovery (Visit) order:
0

Finish order:

Example of a Depth-First Search (cont.)

Choose an adjacent vertex that is not being visited

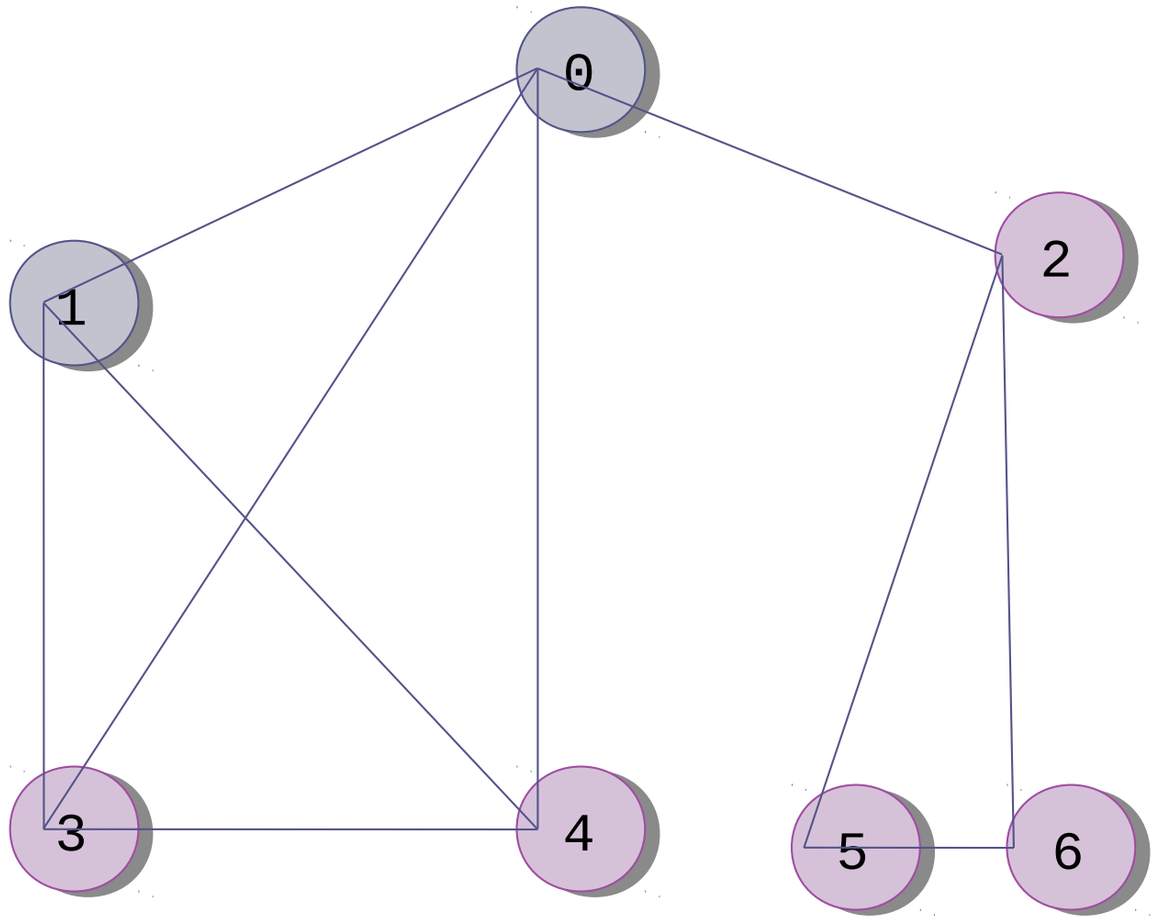


Discovery (Visit) order:
0

Finish order:

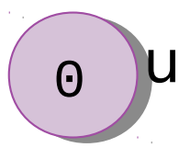
Example of a Depth-First Search (cont.)

Choose an adjacent vertex that is not being visited

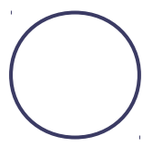


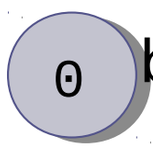
Discovery (Visit) order:
0, 1

Finish order:



unvisited

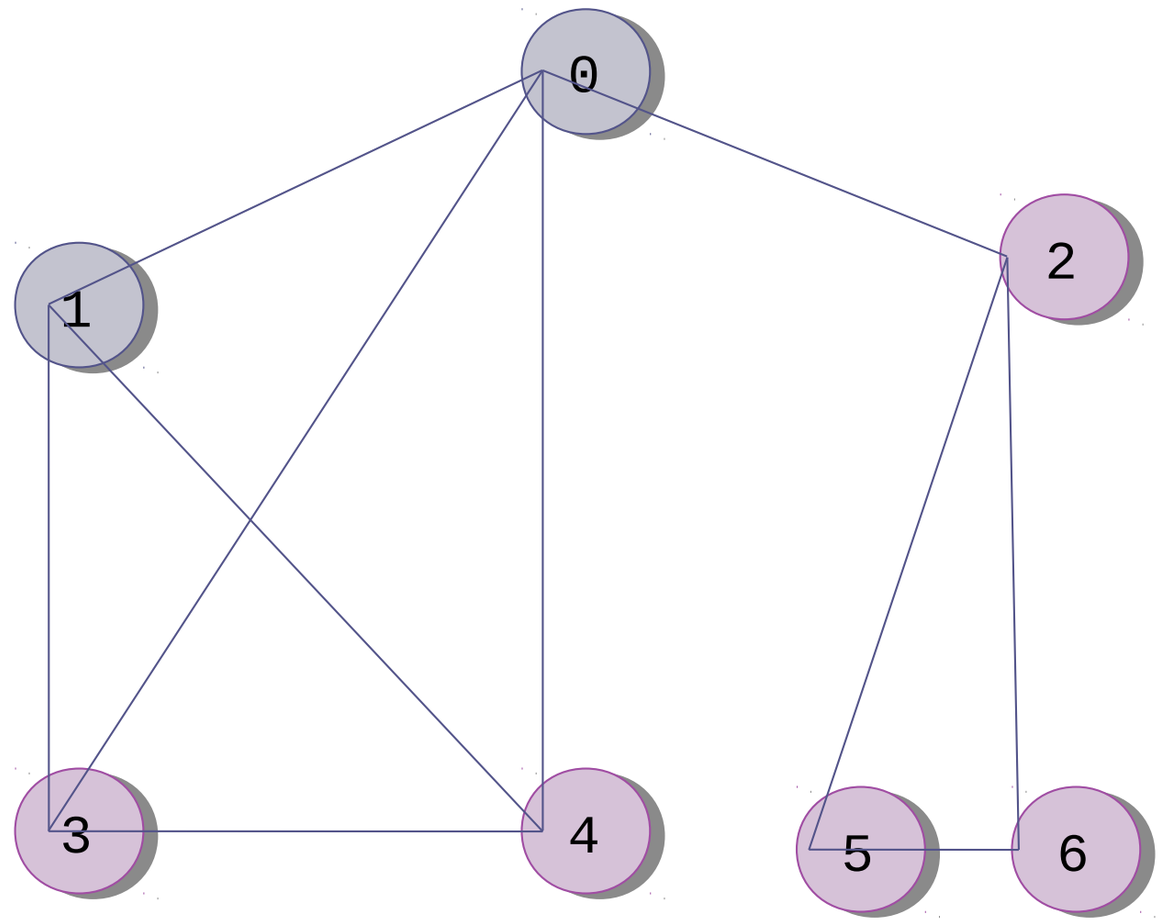

visited



being visited

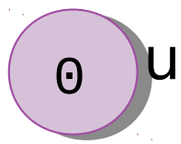
Example of a Depth-First Search (cont.)

(Recursively) choose an adjacent vertex that is not being visited

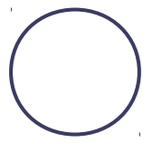


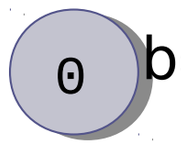
Discovery (Visit) order:
0, 1, 3

Finish order:



unvisited

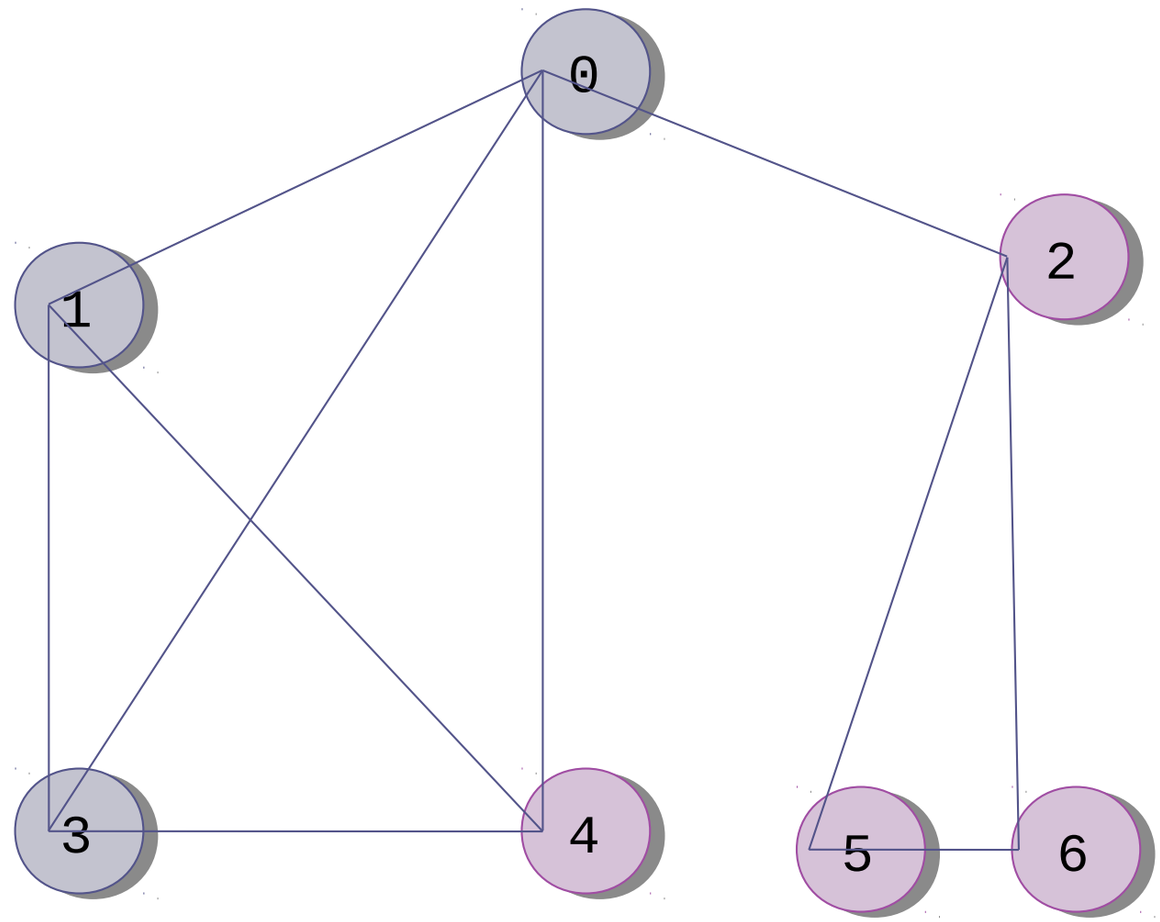

visited



being visited

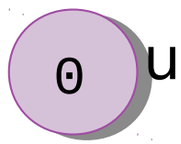
Example of a Depth-First Search (cont.)

(Recursively) choose an adjacent vertex that is not being visited

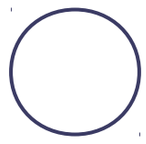


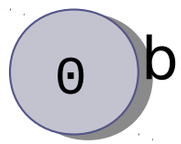
Discovery (Visit) order:
0, 1, 3

Finish order:



unvisited

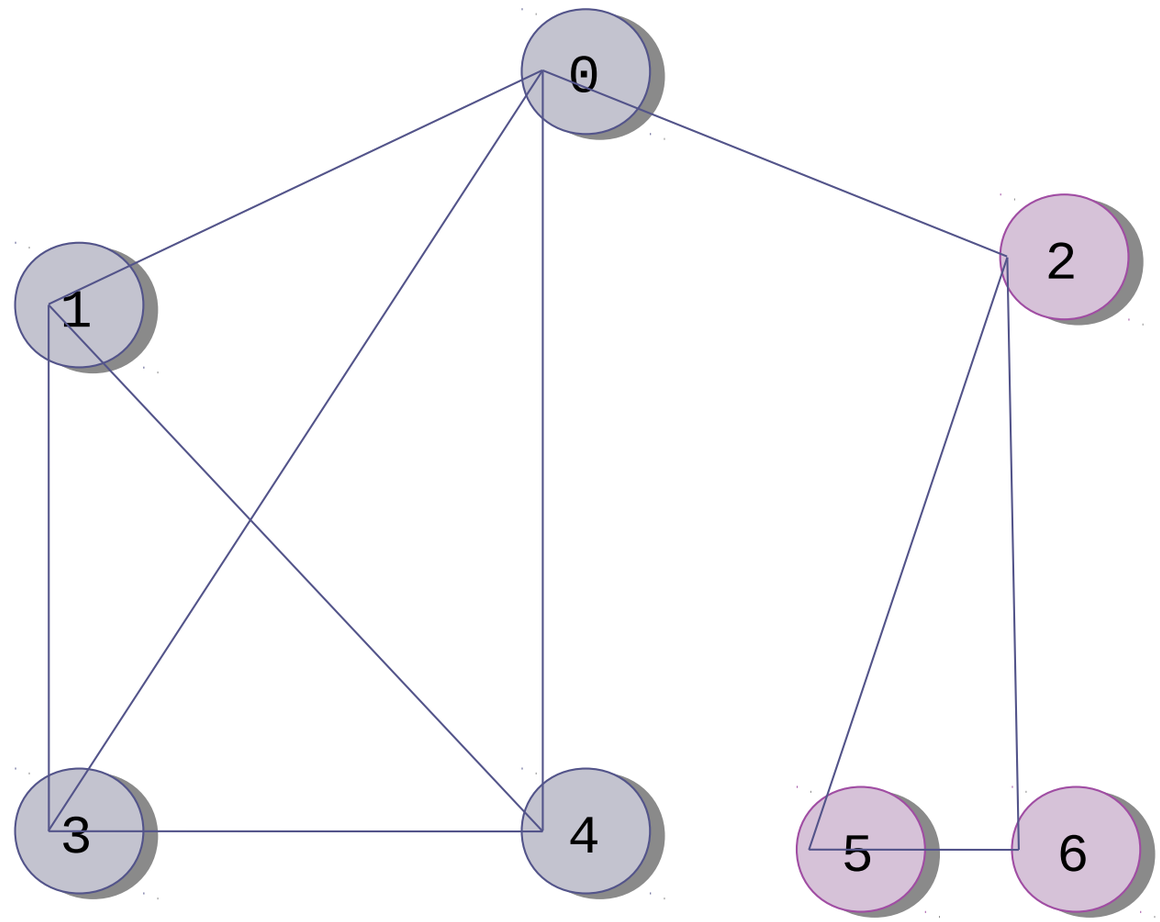

visited



being visited

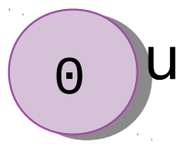
Example of a Depth-First Search (cont.)

(Recursively) choose an adjacent vertex that is not being visited

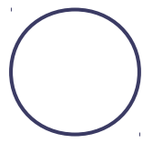


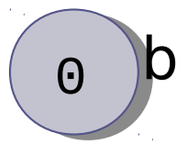
Discovery (Visit) order:
0, 1, 3, 4

Finish order:



unvisited

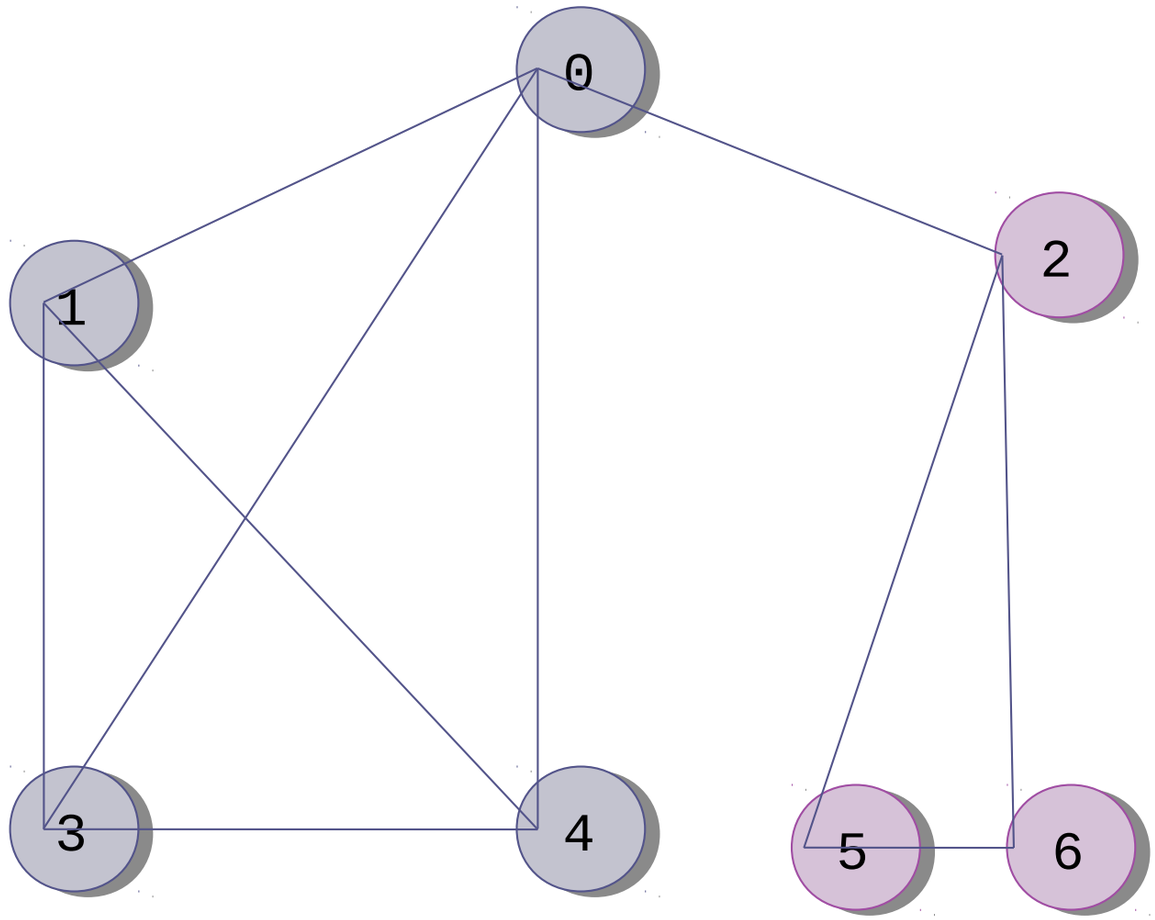

visited



being visited

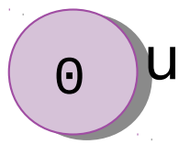
Example of a Depth-First Search (cont.)

There are no vertices adjacent to 4 that are not being visited

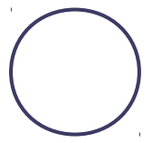


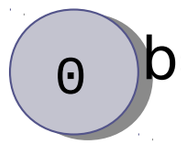
Discovery (Visit) order:
0, 1, 3, 4

Finish order:



unvisited

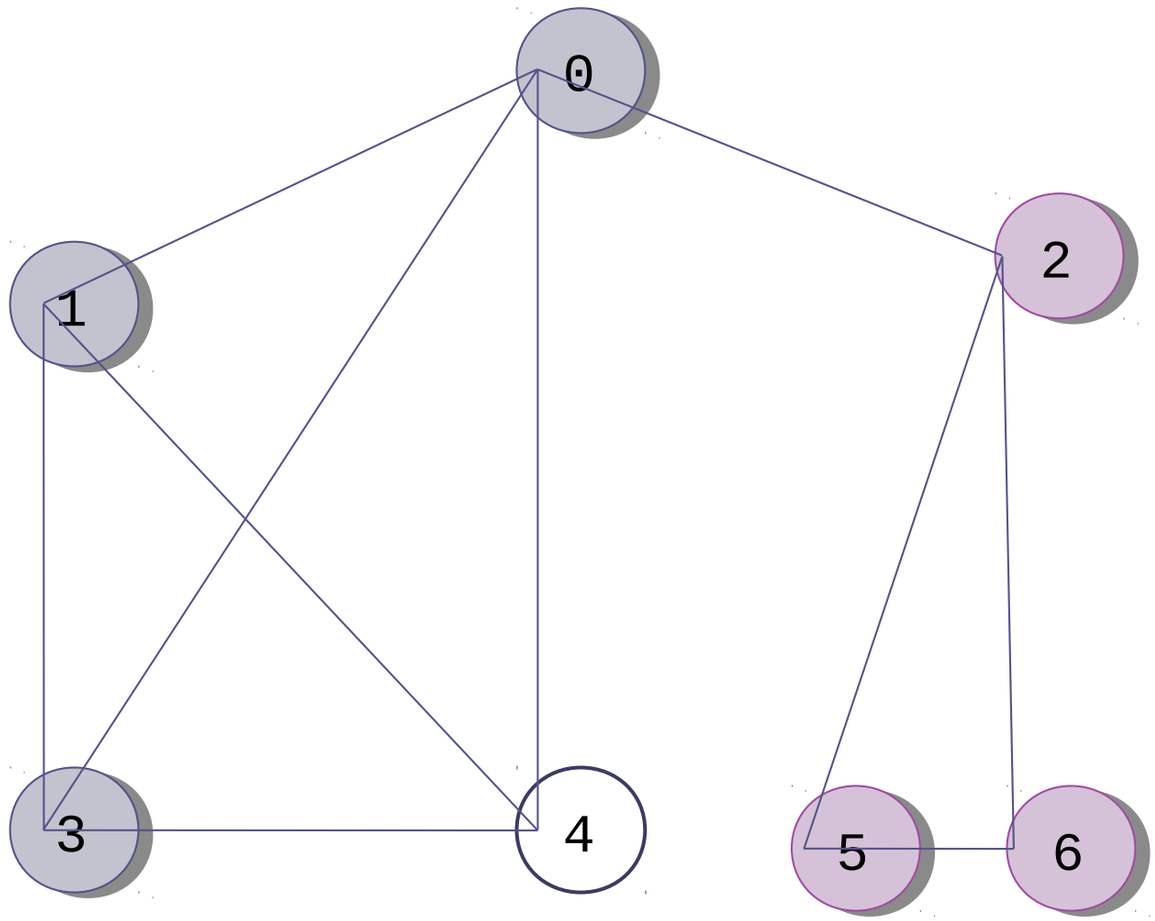

visited



being visited

Example of a Depth-First Search (cont.)

Mark 4 as visited

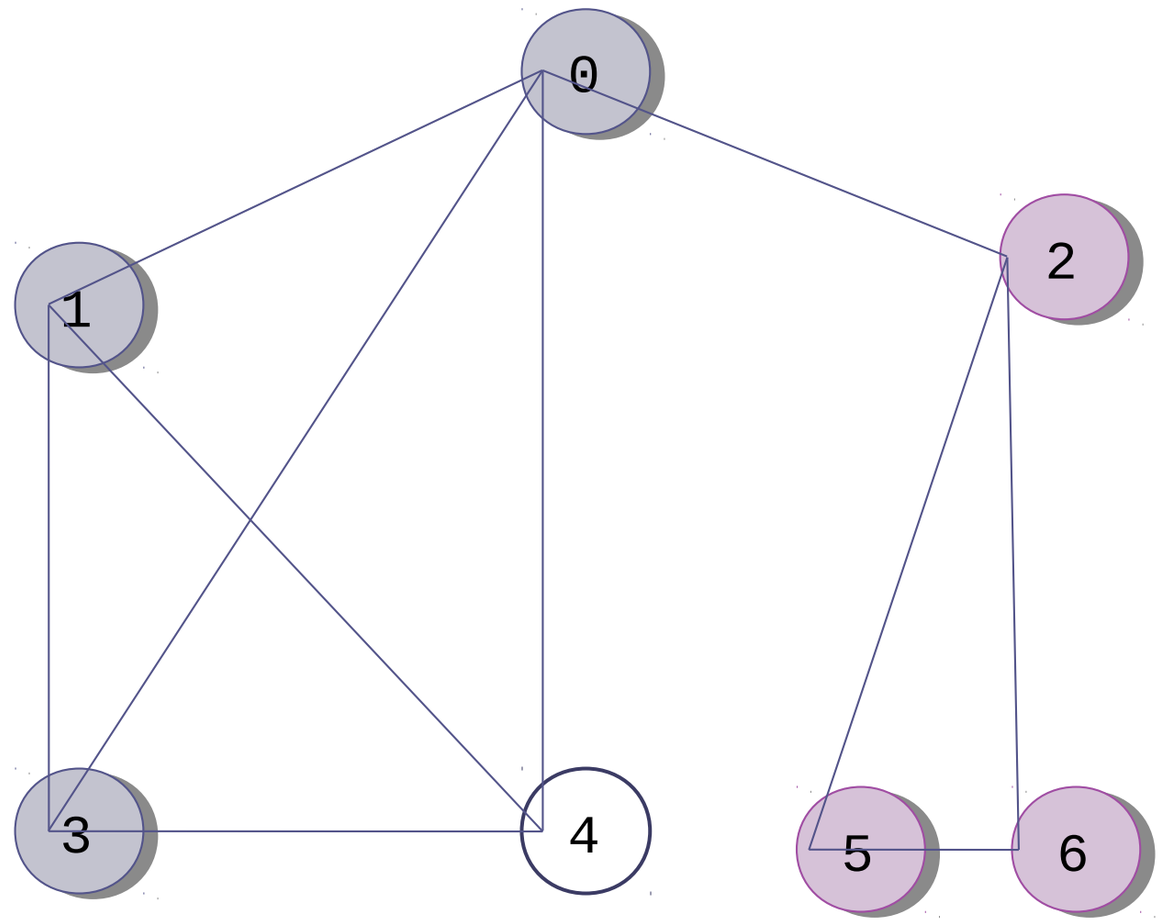


Discovery (Visit) order:
0, 1, 3, 4

Finish order:
4

Example of a Depth-First Search (cont.)

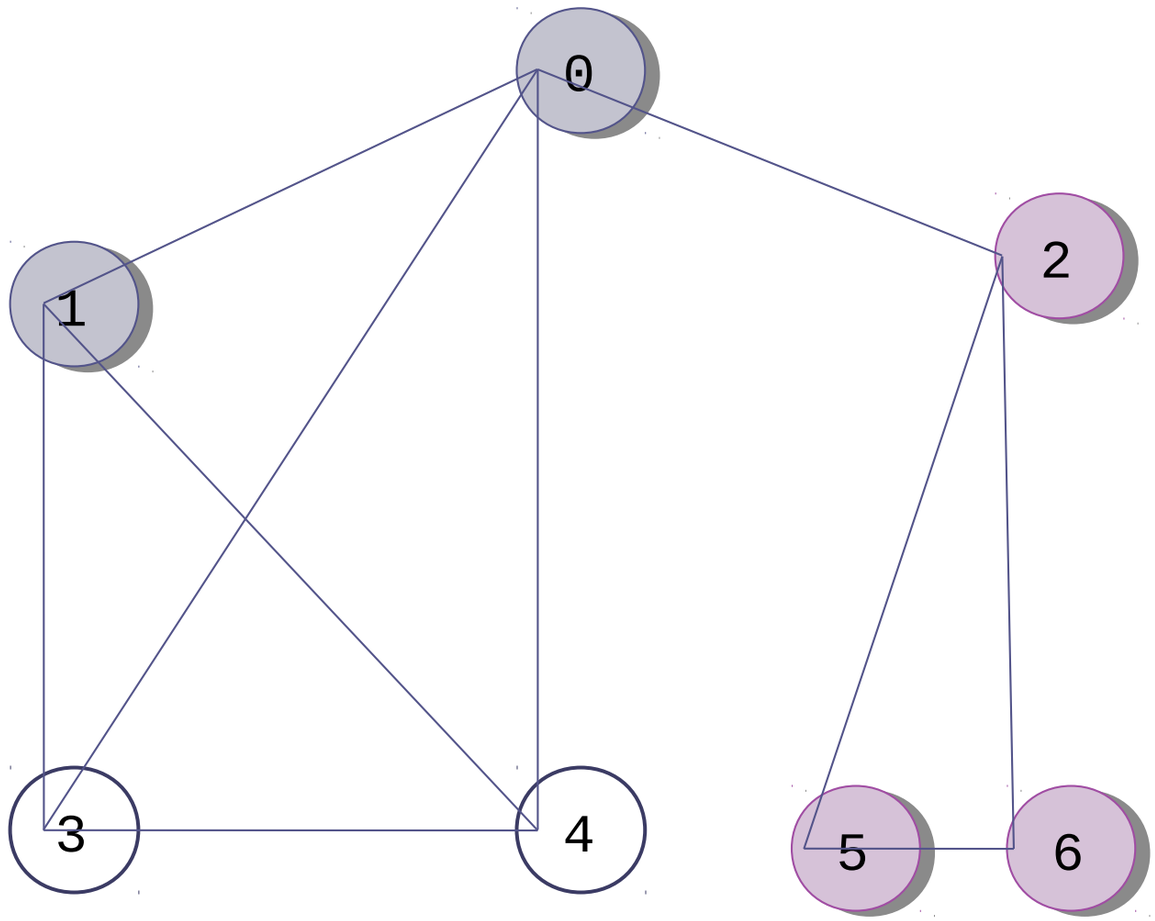
Return from the recursion to 3; all adjacent nodes to 3 are being visited



Finish order:
4

Example of a Depth-First Search (cont.)

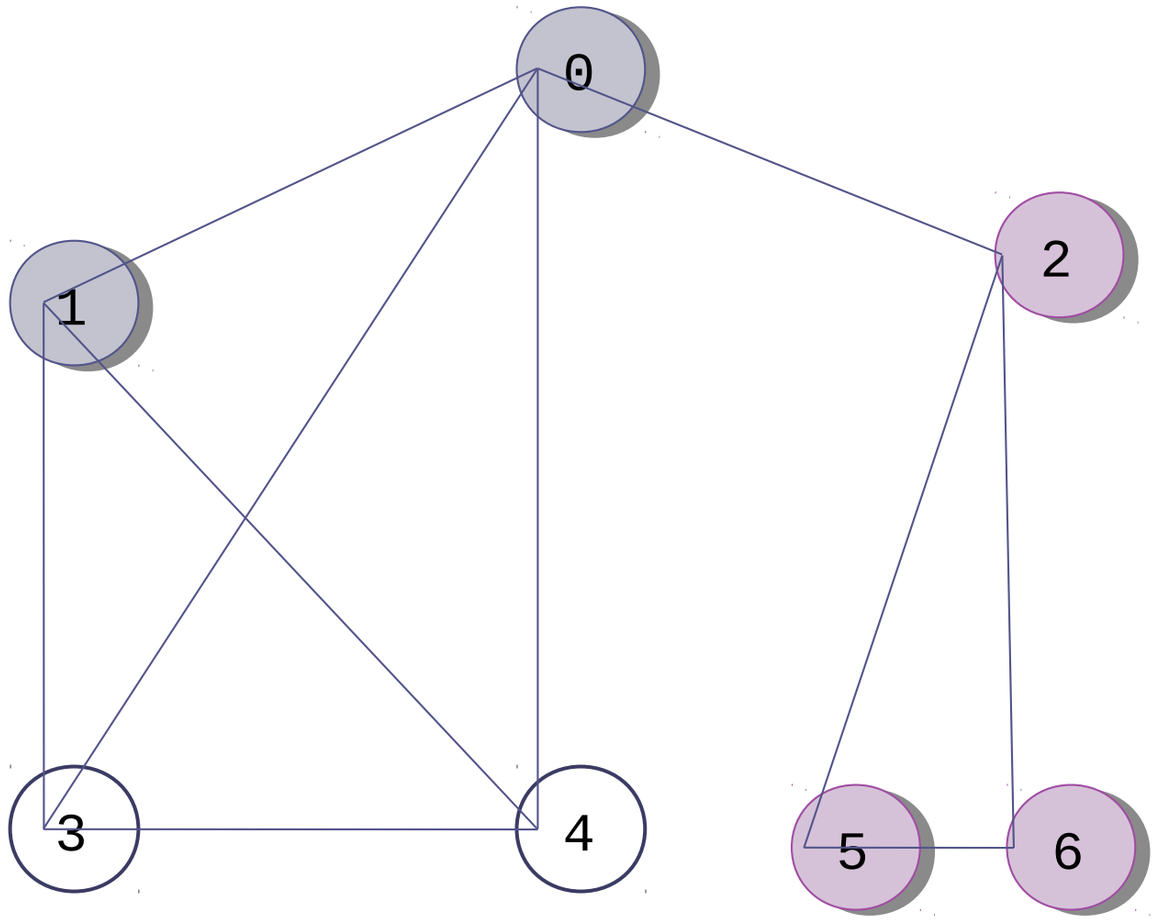
Mark 3 as visited



Finish order:
4, 3

Example of a Depth-First Search (cont.)

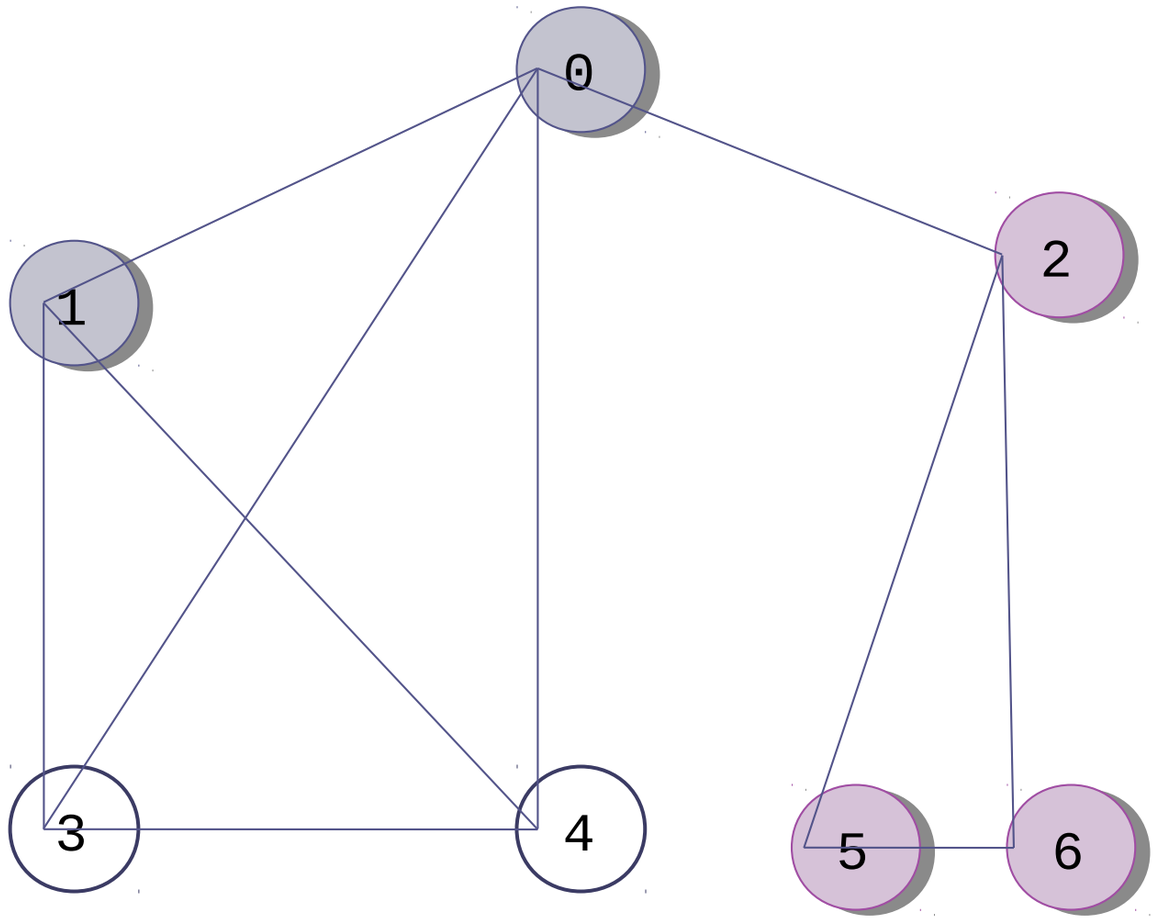
Return from the recursion to 1



Finish order:
4, 3

Example of a Depth-First Search (cont.)

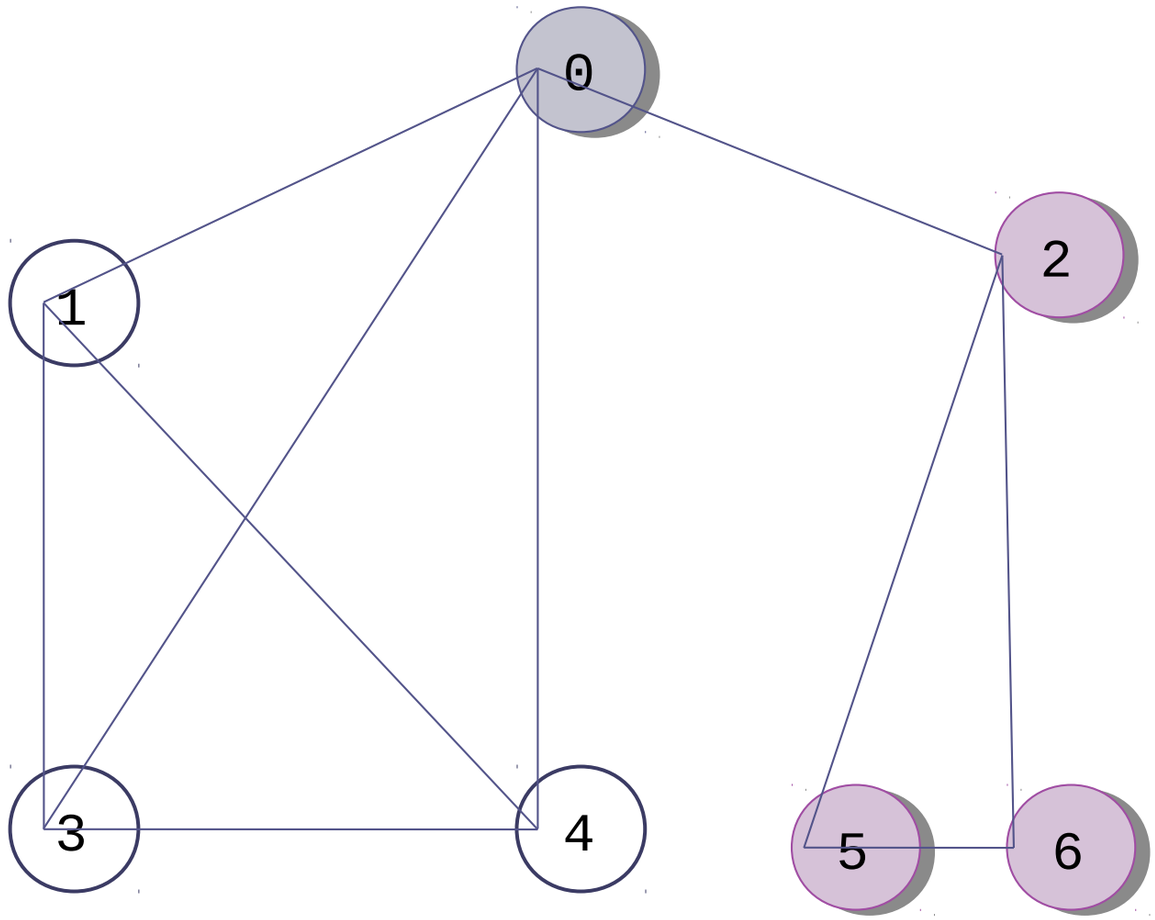
All vertices adjacent to 1 are being visited



Finish order:
4, 3

Example of a Depth-First Search (cont.)

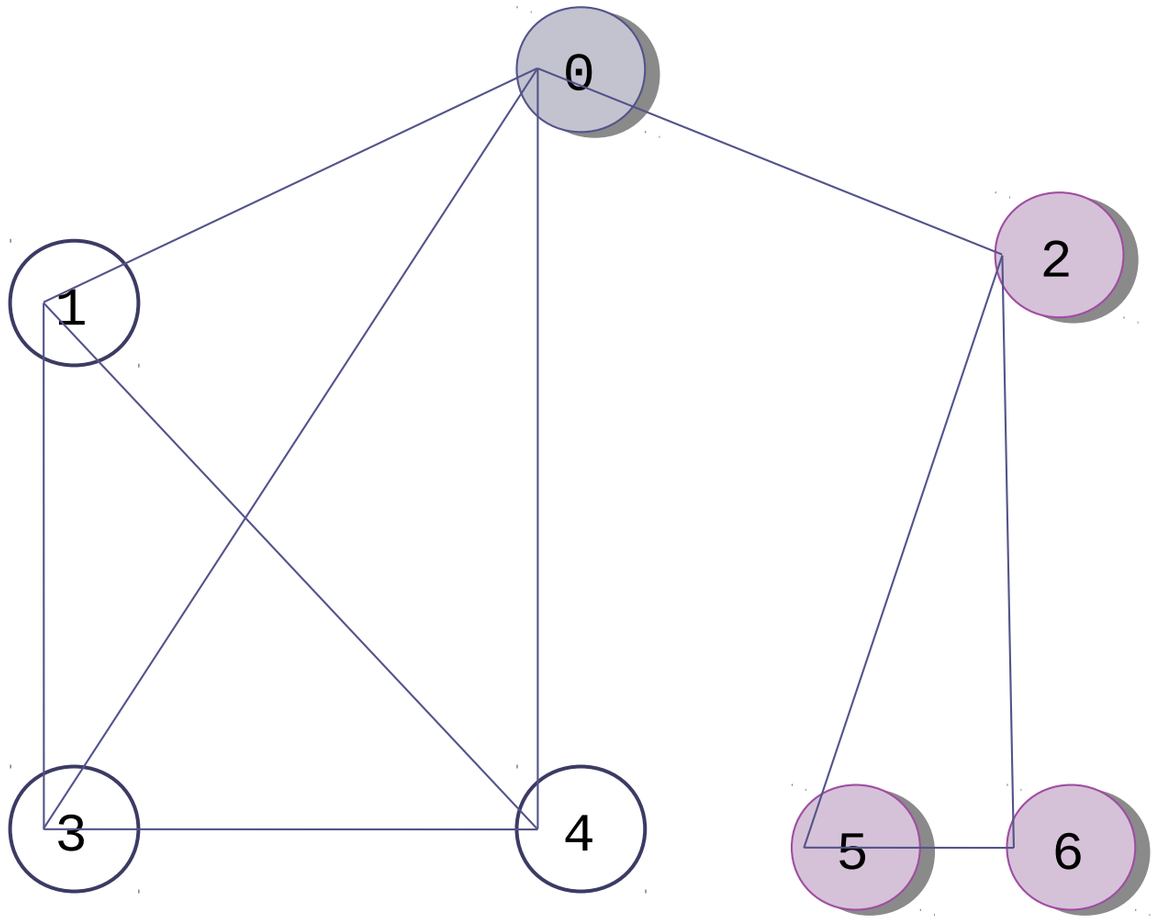
Mark 1 as visited



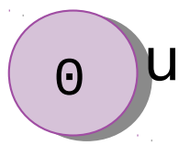
Finish order:
4, 3, 1

Example of a Depth-First Search (cont.)

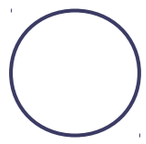
Return from the recursion to 0

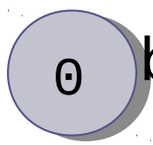


Finish order:
4, 3, 1



unvisited

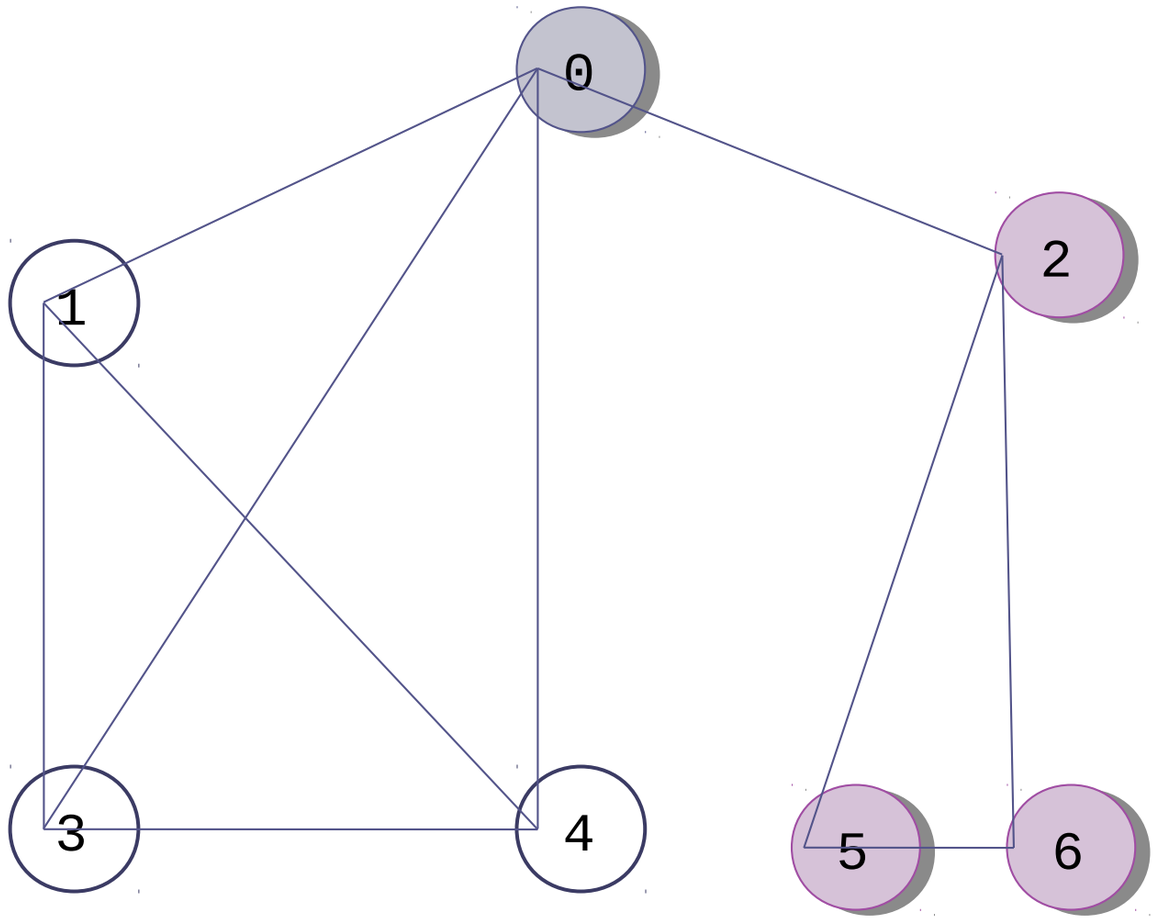

visited



being visited

Example of a Depth-First Search (cont.)

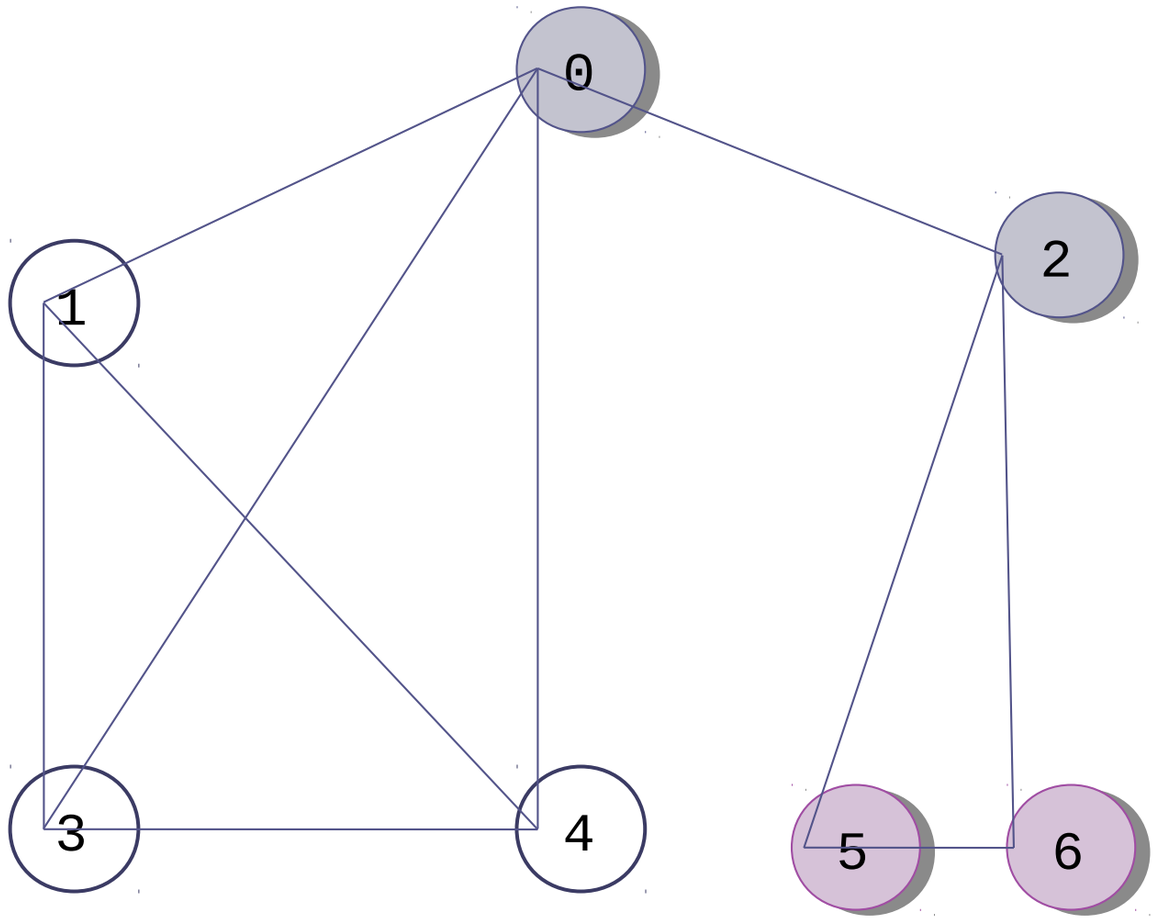
2 is adjacent to 1 and is not being visited



Finish order:
4, 3, 1

Example of a Depth-First Search (cont.)

2 is adjacent to 1 and is not being visited

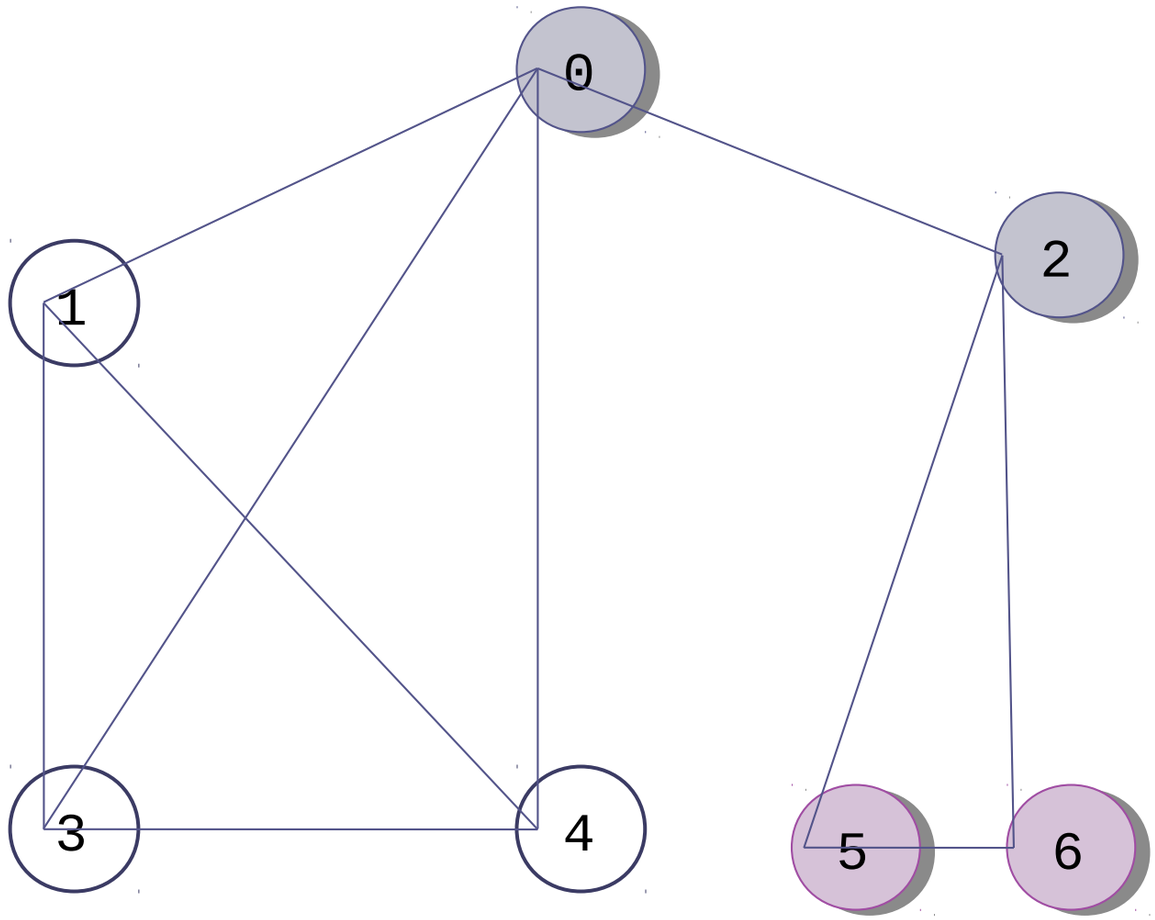


Discovery (Visit) order:
0, 1, 3, 4, 2

Finish order:
4, 3, 1

Example of a Depth-First Search (cont.)

5 is adjacent to 2 and is not being visited

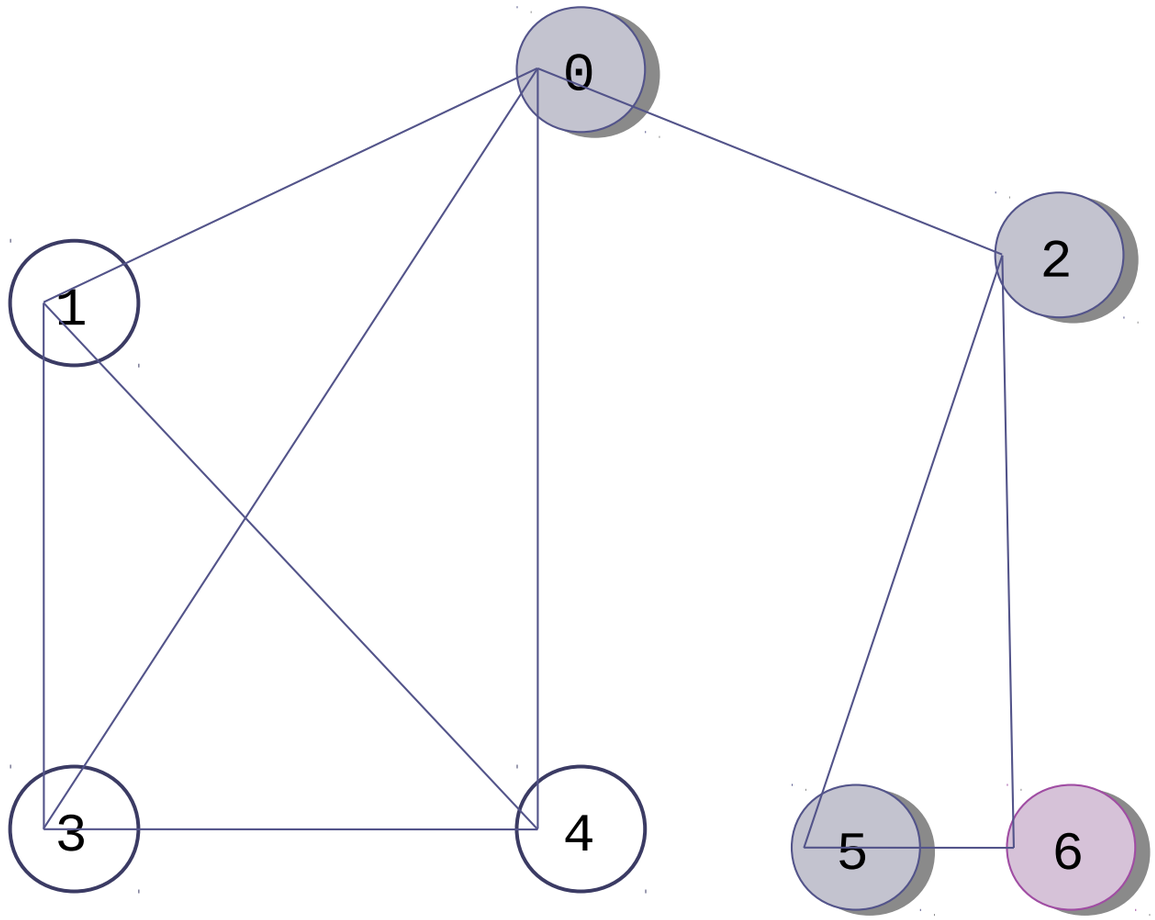


Discovery (Visit) order:
0, 1, 3, 4, 2

Finish order:
4, 3, 1

Example of a Depth-First Search (cont.)

5 is adjacent to 2 and is not being visited

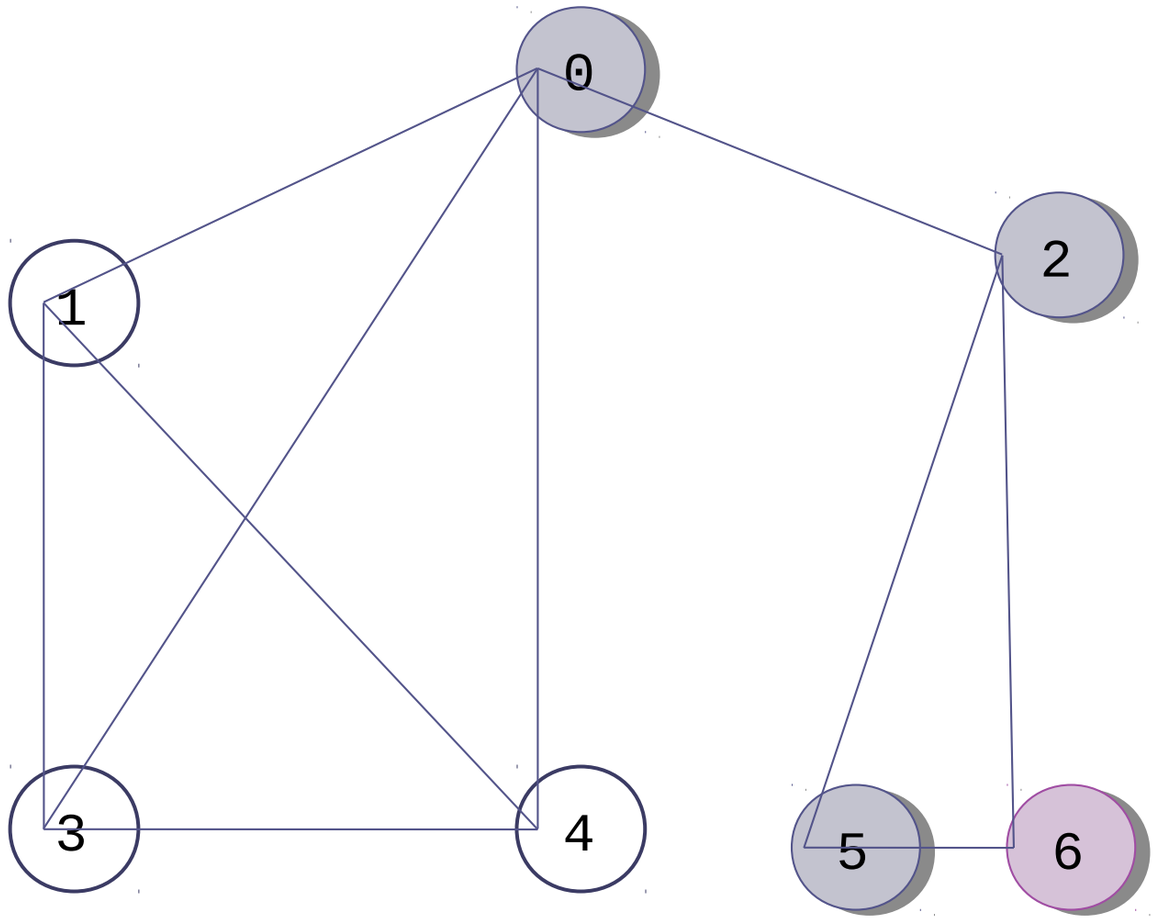


Discovery (Visit) order:
0, 1, 3, 4, 2, 5

Finish order:
4, 3, 1

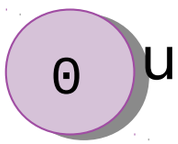
Example of a Depth-First Search (cont.)

6 is adjacent to 5 and is not being visited

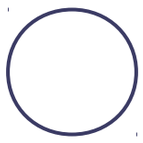


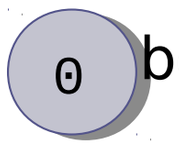
Discovery (Visit) order:
0, 1, 3, 4, 2, 5

Finish order:
4, 3, 1



unvisited

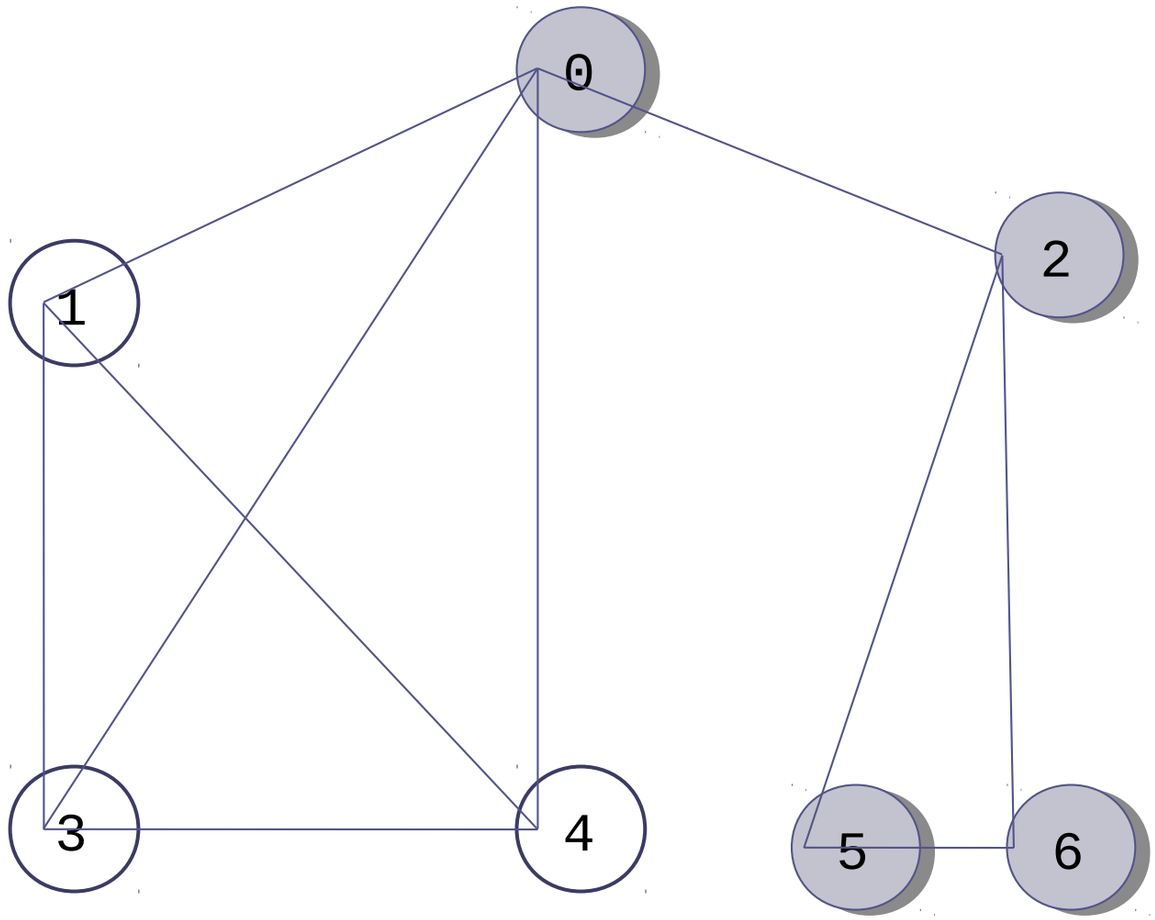

visited



being visited

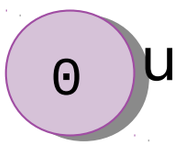
Example of a Depth-First Search (cont.)

6 is adjacent to 5 and is not being visited

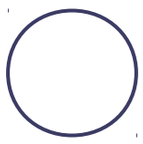


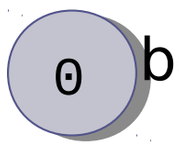
Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6

Finish order:
4, 3, 1



unvisited

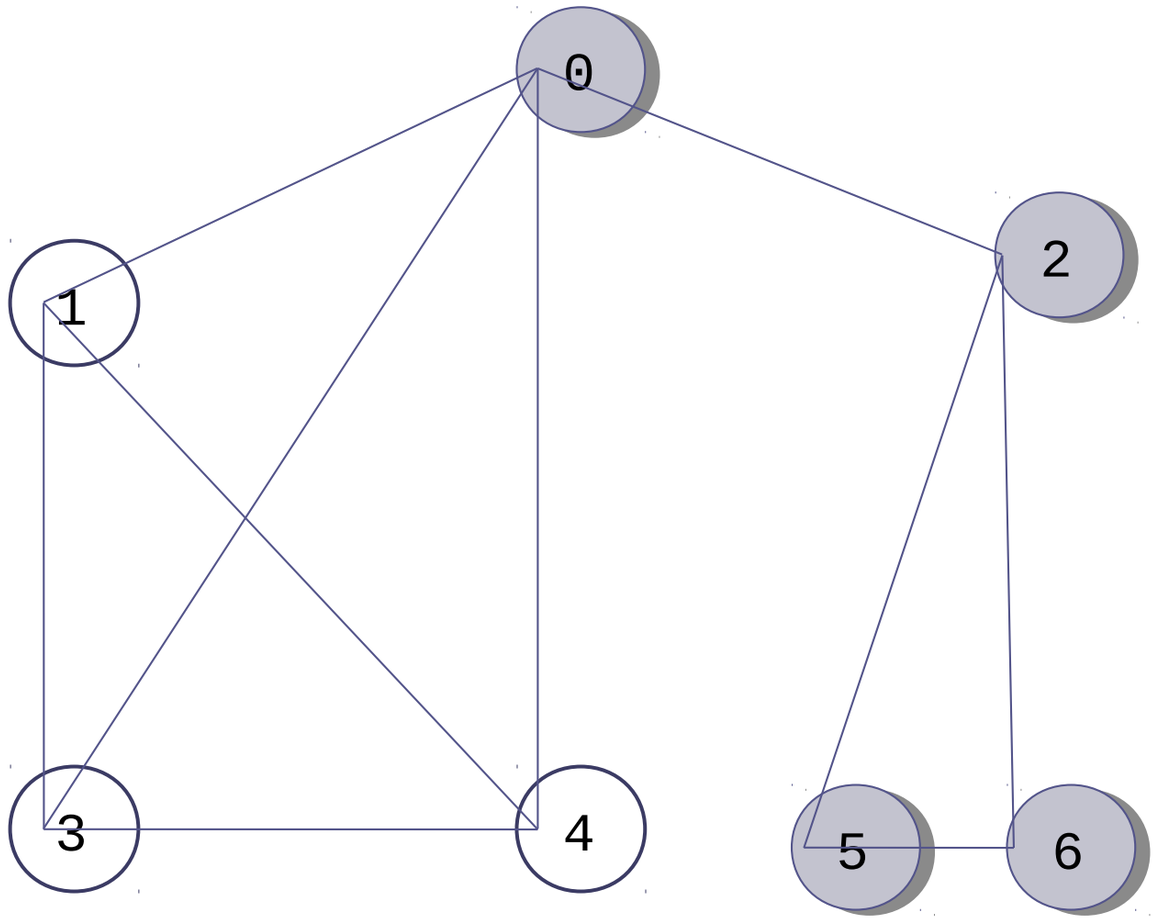

visited



being visited

Example of a Depth-First Search (cont.)

There are no vertices adjacent to 6 not being visited; mark 6 as visited

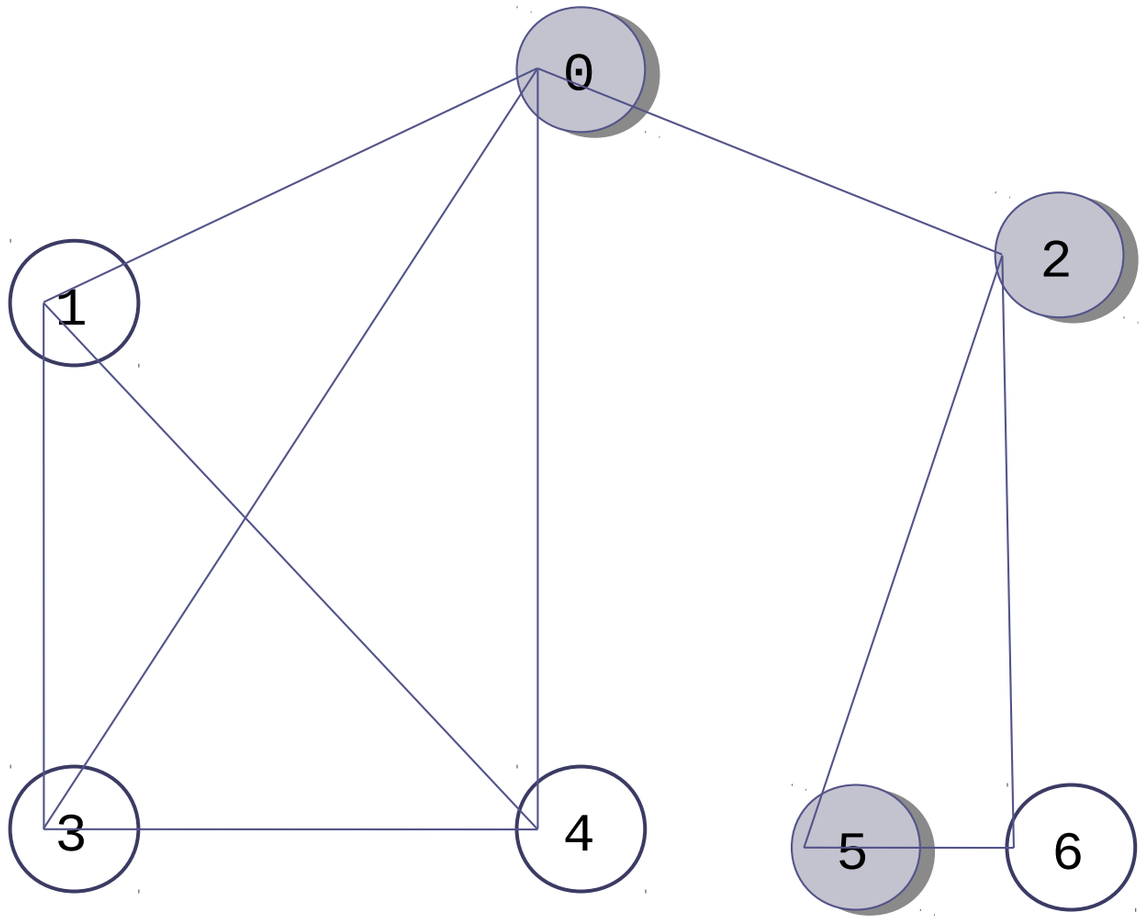


Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6

Finish order:
4, 3, 1

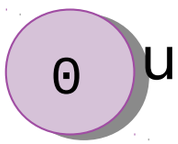
Example of a Depth-First Search (cont.)

There are no vertices adjacent to 6 not being visited; mark 6 as visited

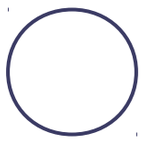


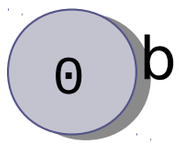
Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6

Finish order:
4, 3, 1, 6



unvisited

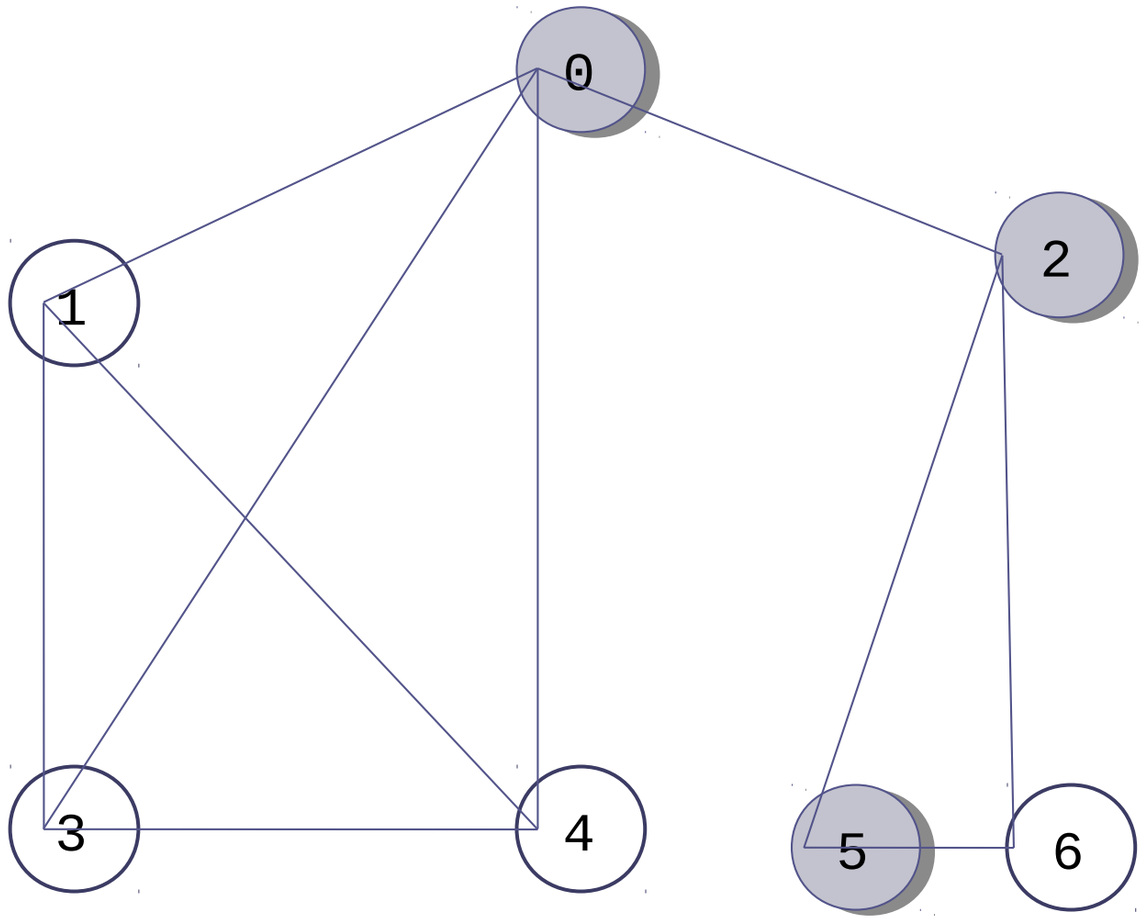

visited



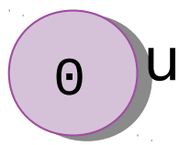
being visited

Example of a Depth-First Search (cont.)

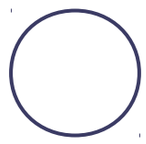
Return from the recursion to 5

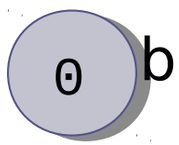


Finish order:
4, 3, 1, 6



unvisited

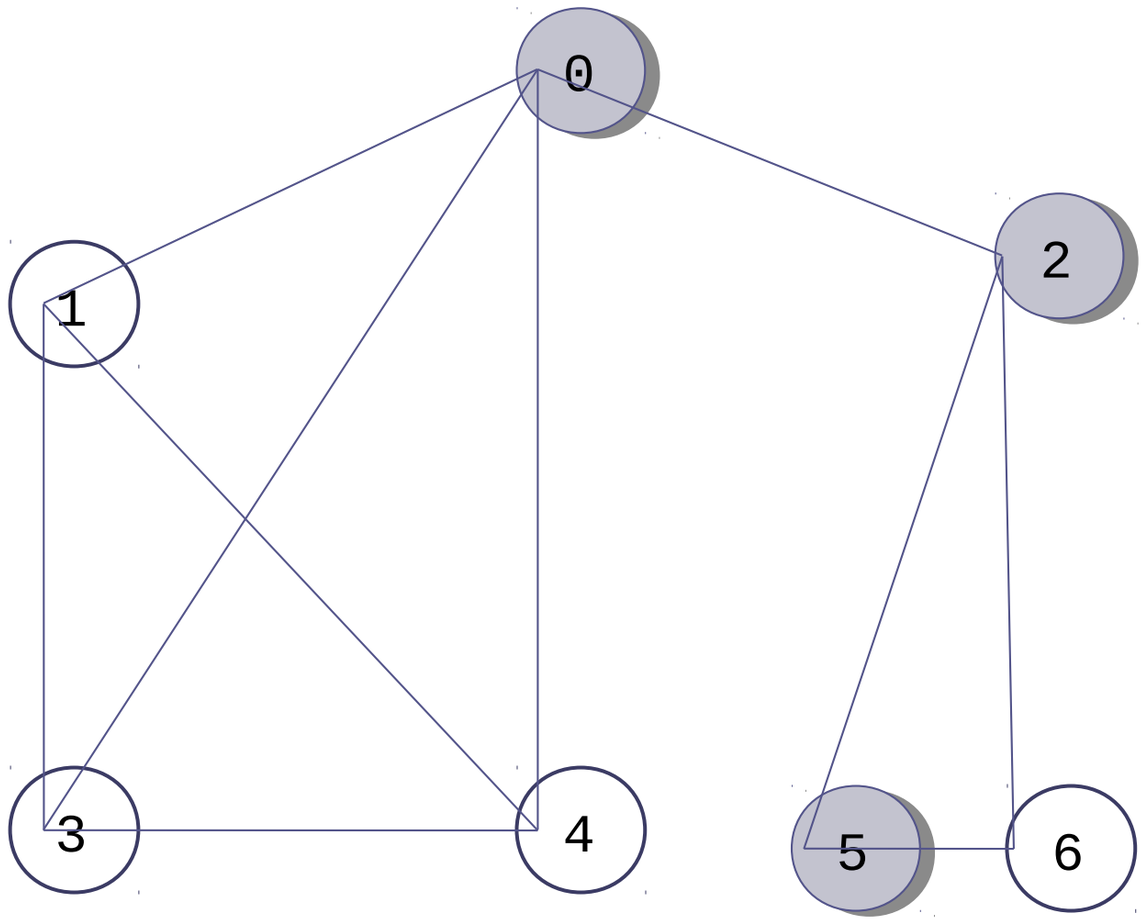

visited



being visited

Example of a Depth-First Search (cont.)

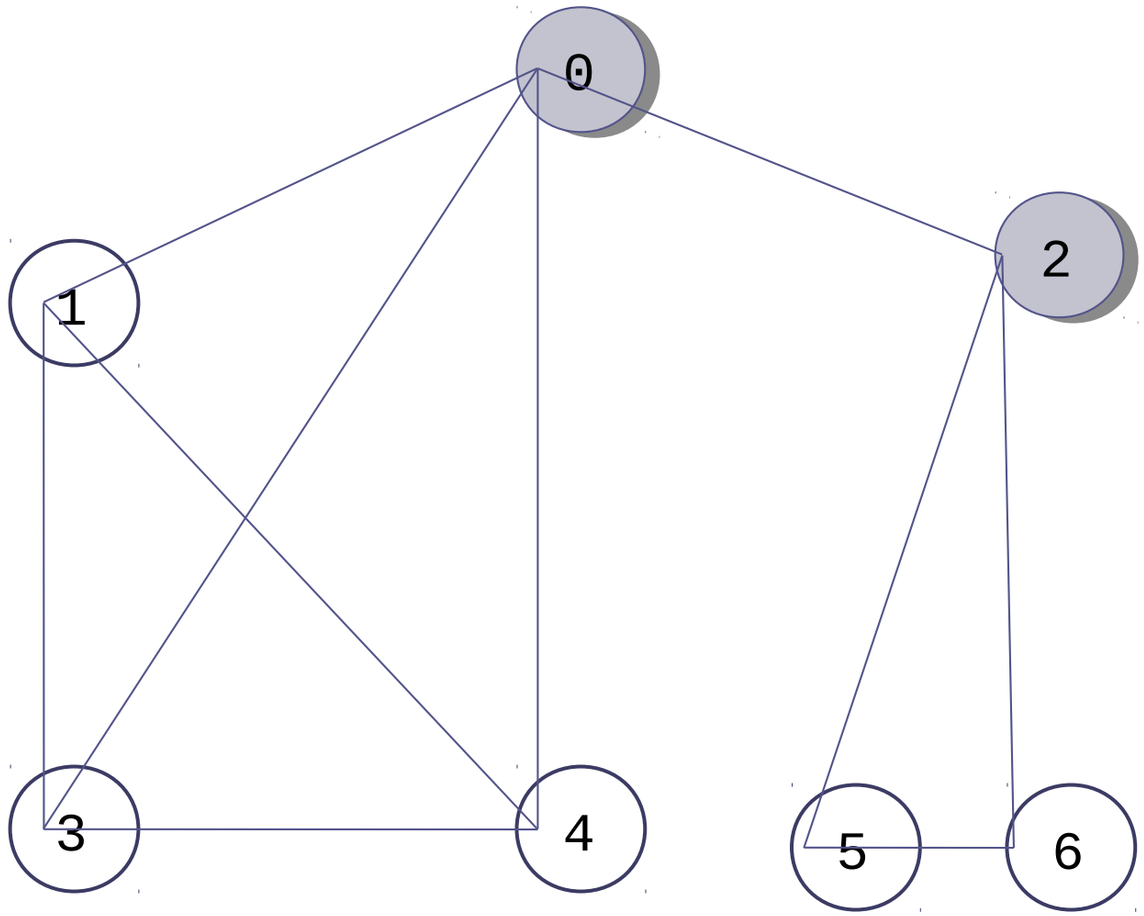
Mark 5 as visited



Finish order:
4, 3, 1, 6

Example of a Depth-First Search (cont.)

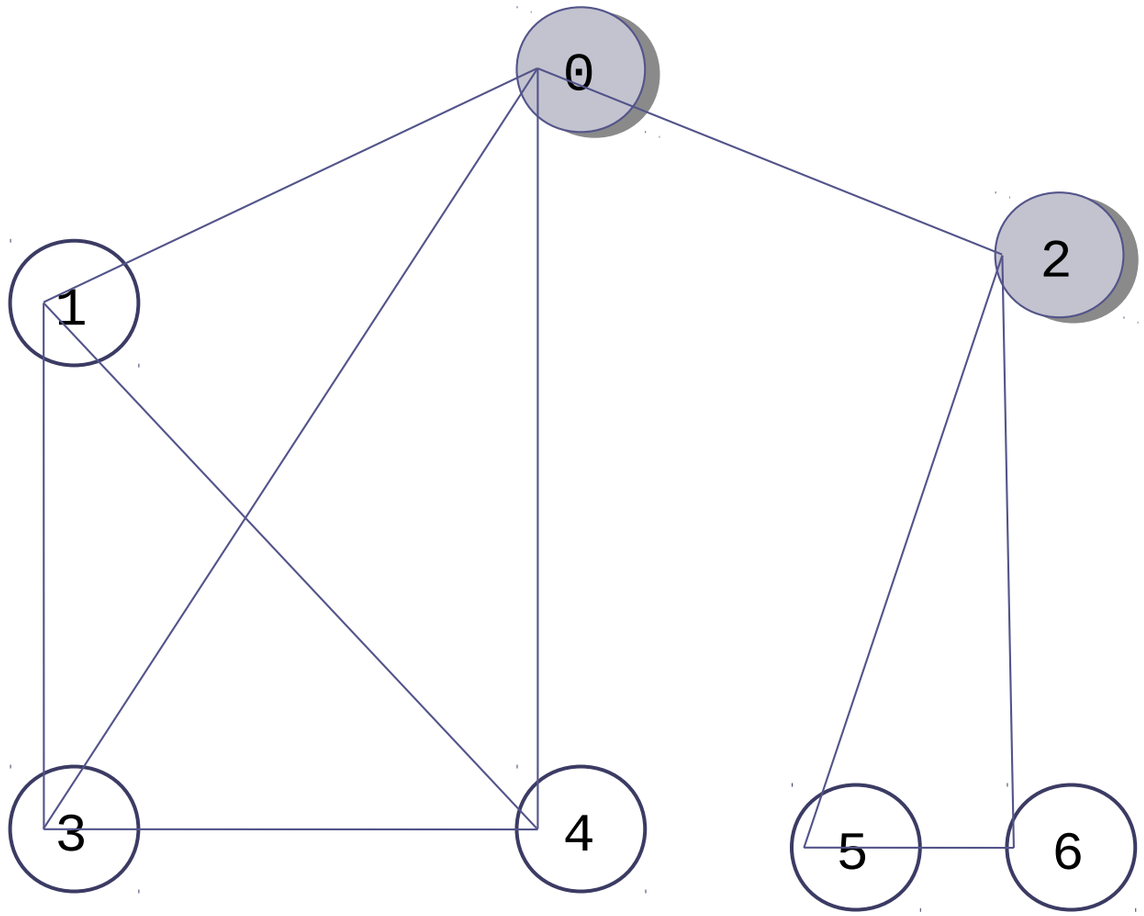
Mark 5 as visited



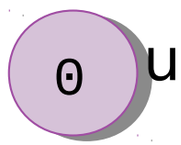
Finish order:
4, 3, 1, 6, 5

Example of a Depth-First Search (cont.)

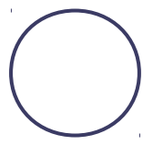
Return from the recursion to 2

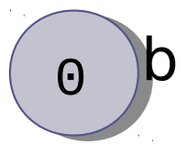


Finish order:
4, 3, 1, 6, 5



unvisited

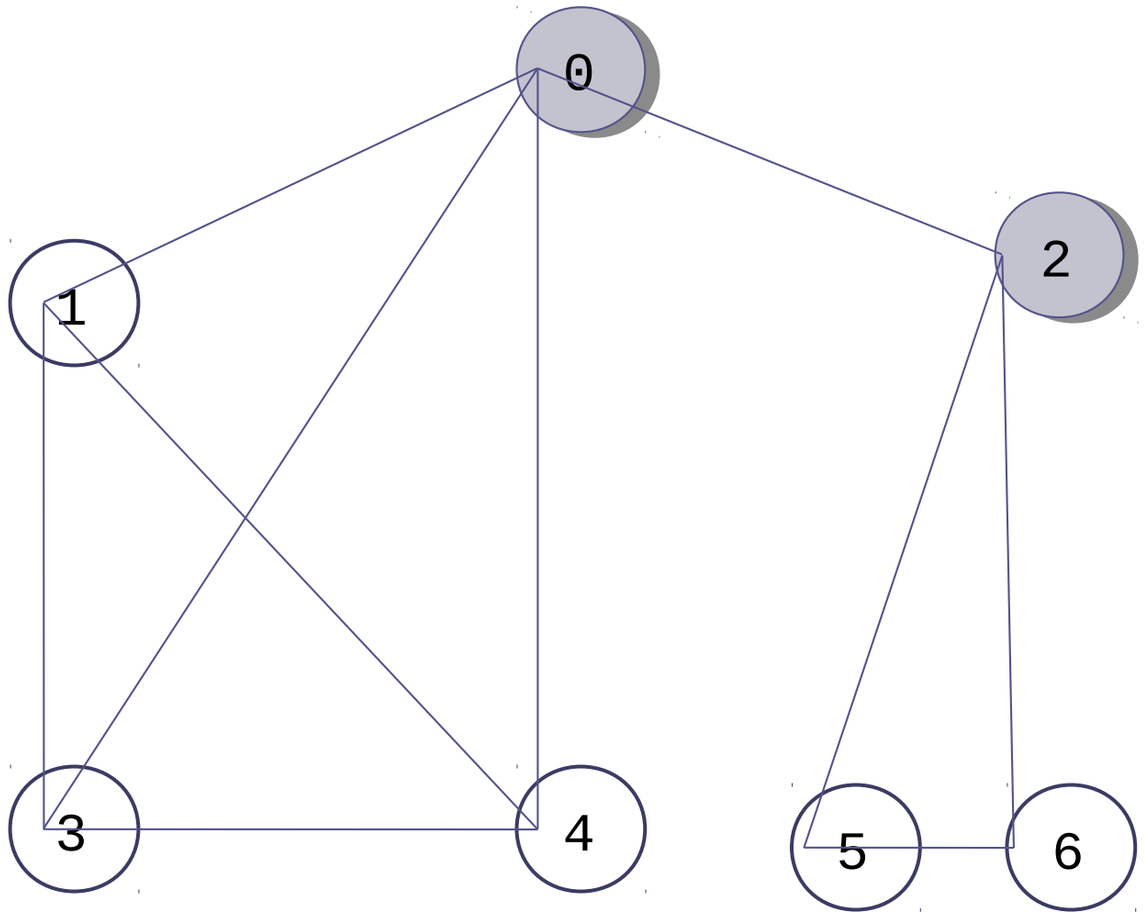

visited



being visited

Example of a Depth-First Search (cont.)

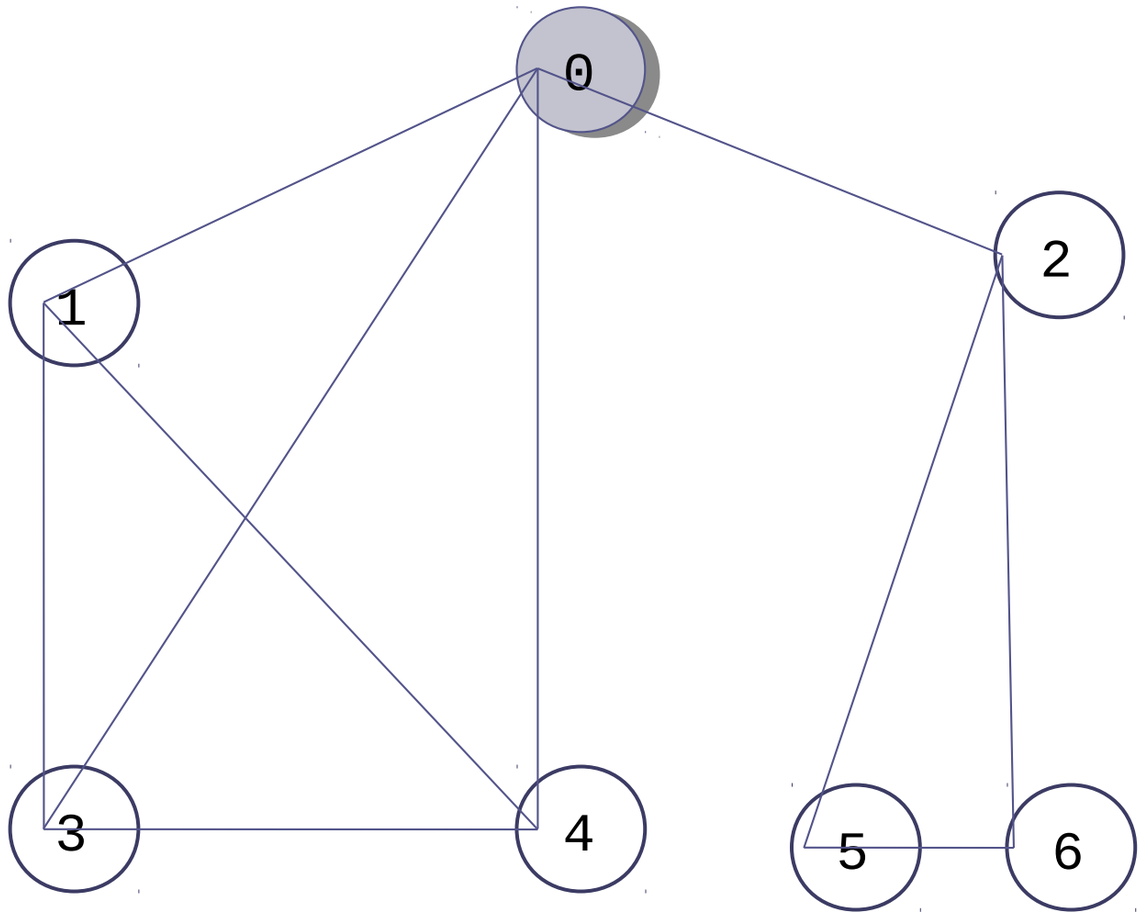
Mark 2 as visited



Finish order:
4, 3, 1, 6, 5

Example of a Depth-First Search (cont.)

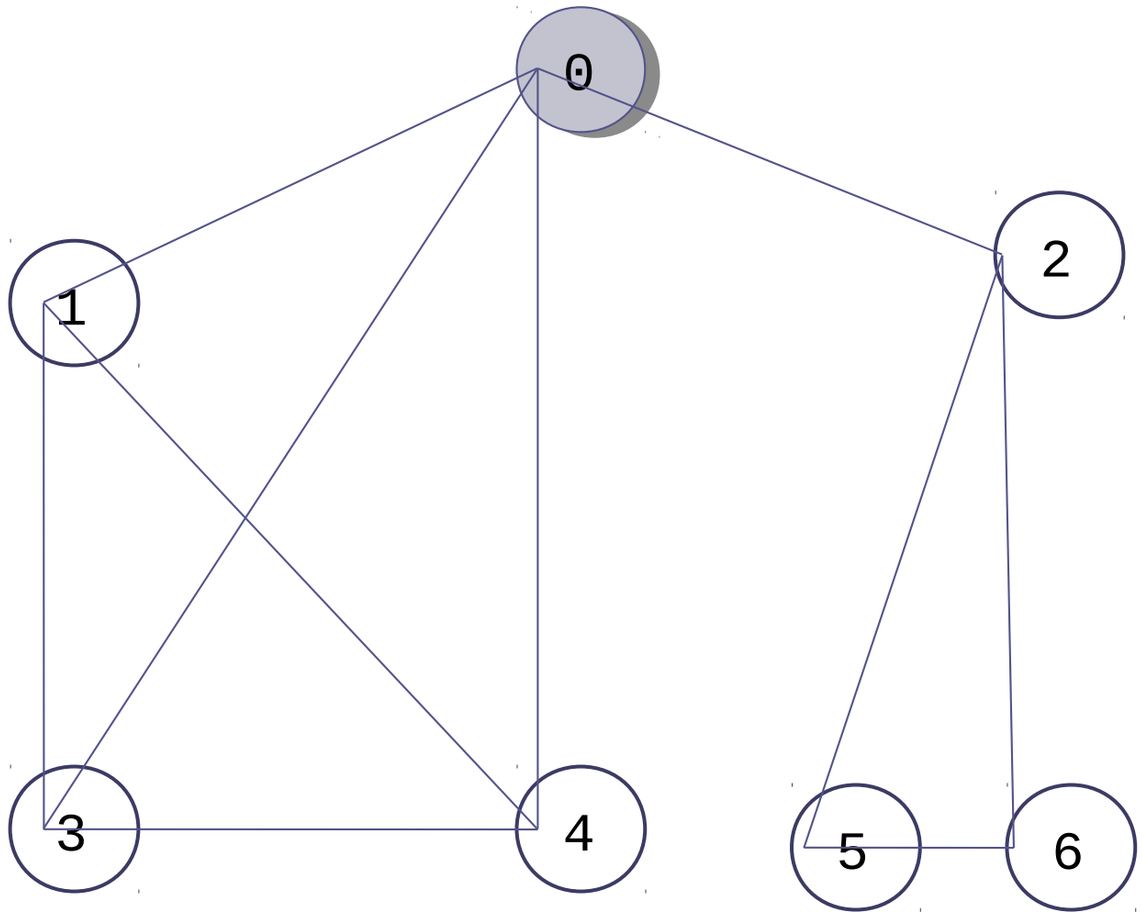
Mark 2 as visited



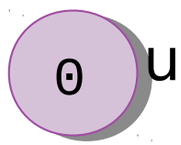
Finish order:
4, 3, 1, 6, 5, 2

Example of a Depth-First Search (cont.)

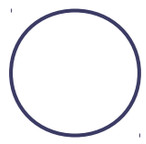
Return from the recursion to 0

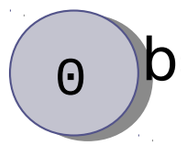


Finish order:
4, 3, 1, 6, 5, 2



unvisited

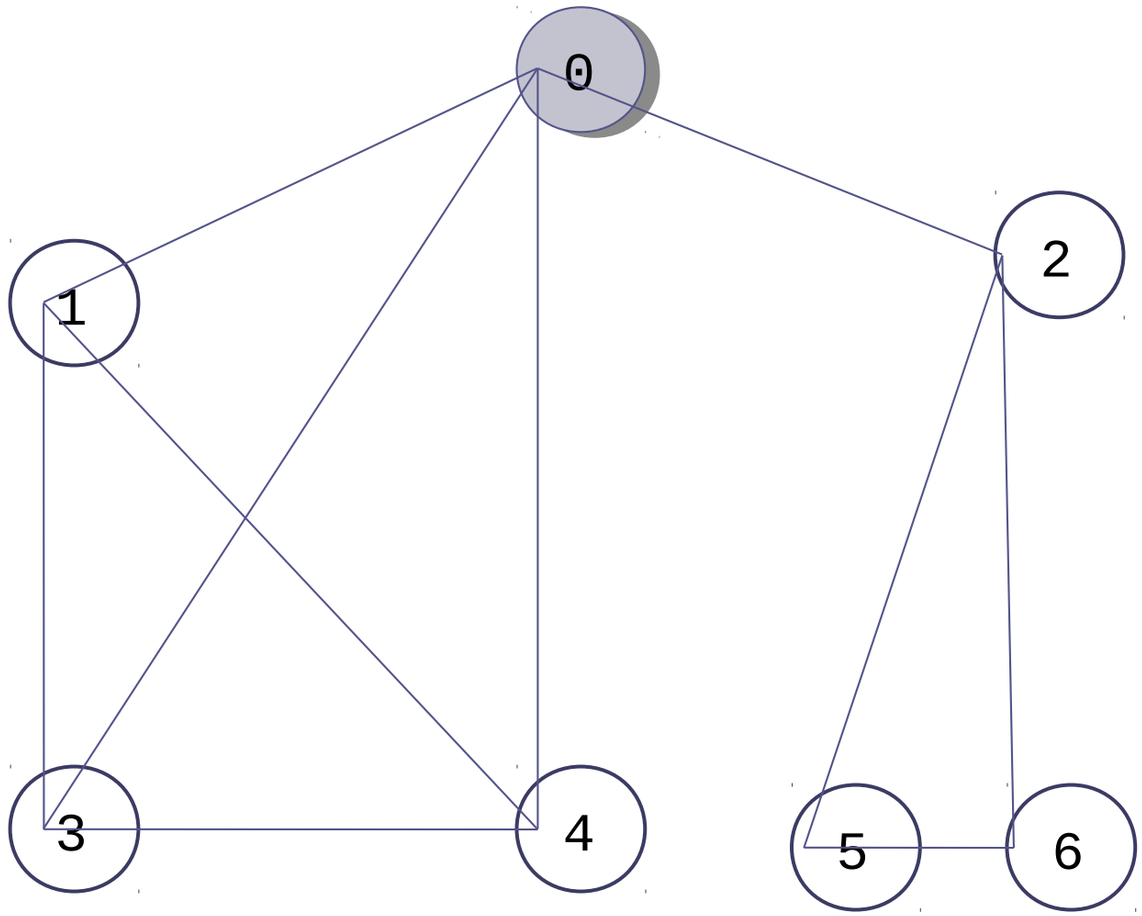

visited



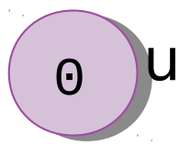
being visited

Example of a Depth-First Search (cont.)

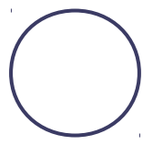
There are no nodes adjacent to 0 not being visited

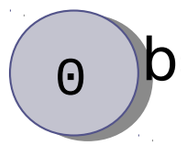


Finish order:
4, 3, 1, 6, 5, 2



unvisited

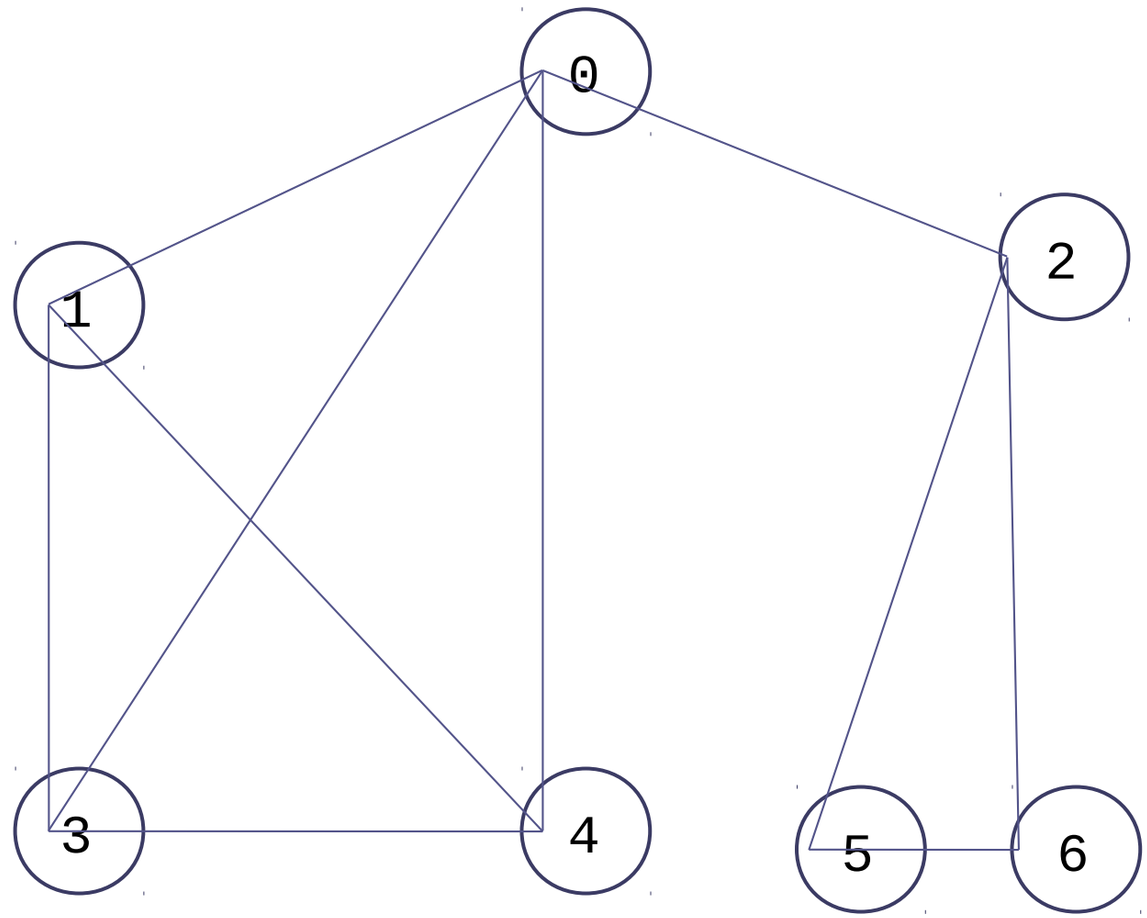

visited



being visited

Example of a Depth-First Search (cont.)

Mark 0 as visited



Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6, 0

Finish order:
4, 3, 1, 6, 5, 2, 0

Upptäcktsordning

Upptäcktsordningen (discovery order) är den ordning vilken noderna upptäcks:

- 0, 1, 3, 4, 2, 5, 6 i detta exempel

Avslutningsordningen (finish order) är ordningen i vilken noderna avslutas

- 4, 3, 1, 6, 5, 2, 0 i detta exempel
- vi kan lagra bakåtpekare, precis som vi gjorde för bredden-först

Algoritm för bredden-först djupet

Algorithm for Depth -First Search

1. Take an arbitrary start vertex, mark it identified (color it light blue), and place it in a stack
2. while the stack is not empty
3. Take a vertex, u, out of the stack and visit u.
4. for all vertices, v, adjacent to this vertex, u
5. if v has not been identified or visited
6. Mark it identified (color it light blue).
7. Insert vertex v into the stack
8. We are now finished visiting u (color it dark blue).

Djupet-först kan implementeras på exakt samma sätt som bredden-först, fast med en stack istället för en kö

Djupet-först, rekursivt

Algorithm for Depth-First Search

1. Mark the current vertex, *u*, visited (color it light blue), and enter it in the discovery order list
2. for each vertex, *v*, adjacent to the current vertex, *u*
3. if *v* has not been visited
4. Set parent of *v* to *u*.
5. Recursively apply this algorithm starting at *v*.
6. Mark *u* finished (color it dark blue) and enter *u* into the finish order list.

Men det går lika bra med en rekursiv implementation, eftersom det är ett sätt att ”dölja” att man använder en stack...

Dessutom kan vi spara avslutningsordningen enkelt
(finish order)

Komplexitet för BFS/DFS

Varje båge testas maximalt en gång

- (två gånger för oriktade grafer)

I värsta fallet blir det alltså |E| tester

- ($2 \cdot |E|$ för oriktade grafer)

Dvs, komplexiteten är O(|E|)

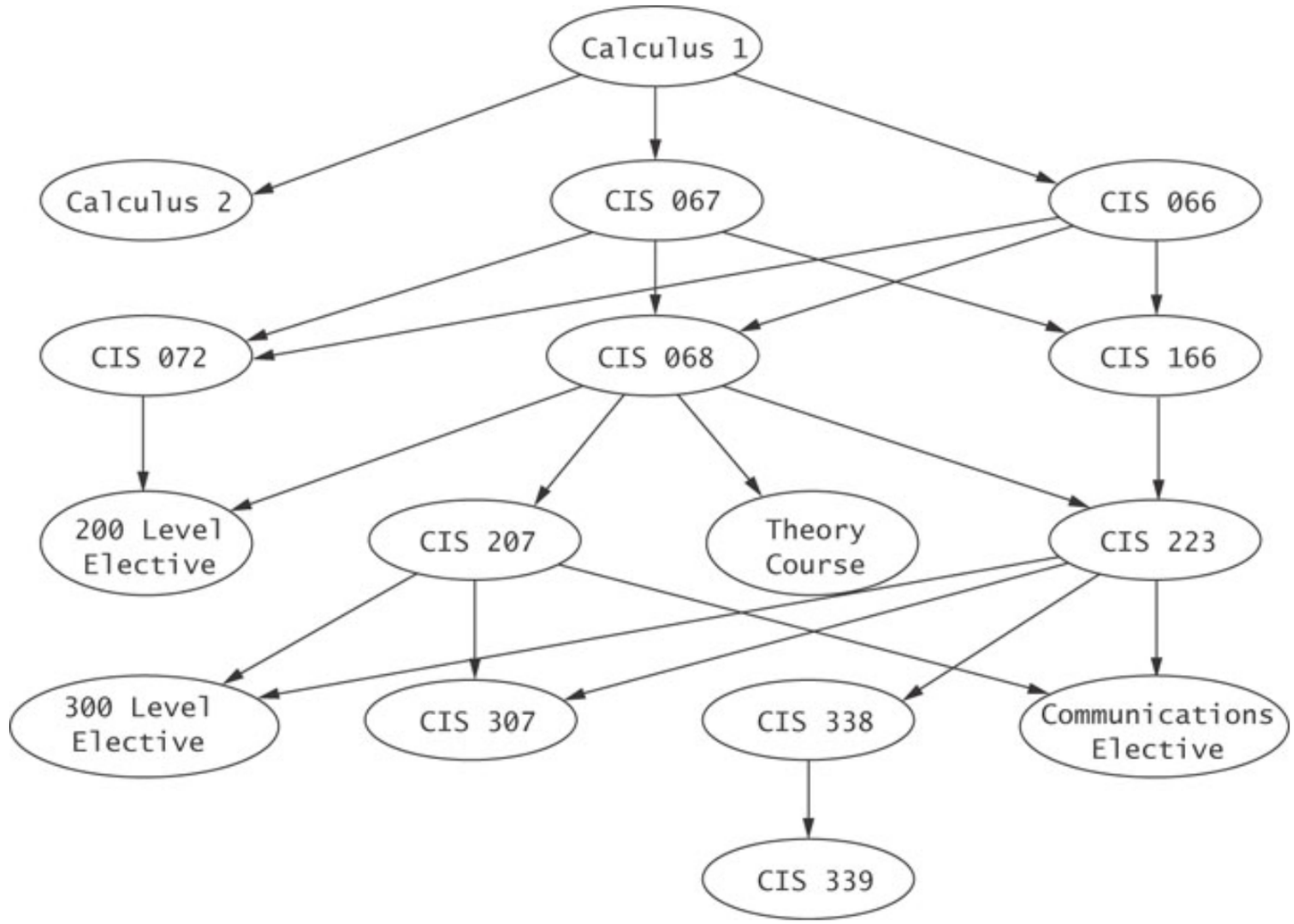
- (även för oriktade grafer)

Detta gäller både bredden-först och djupet-först

DAG: Riktade acykliska grafer

Detta är en DAG (directed acyclic graph)

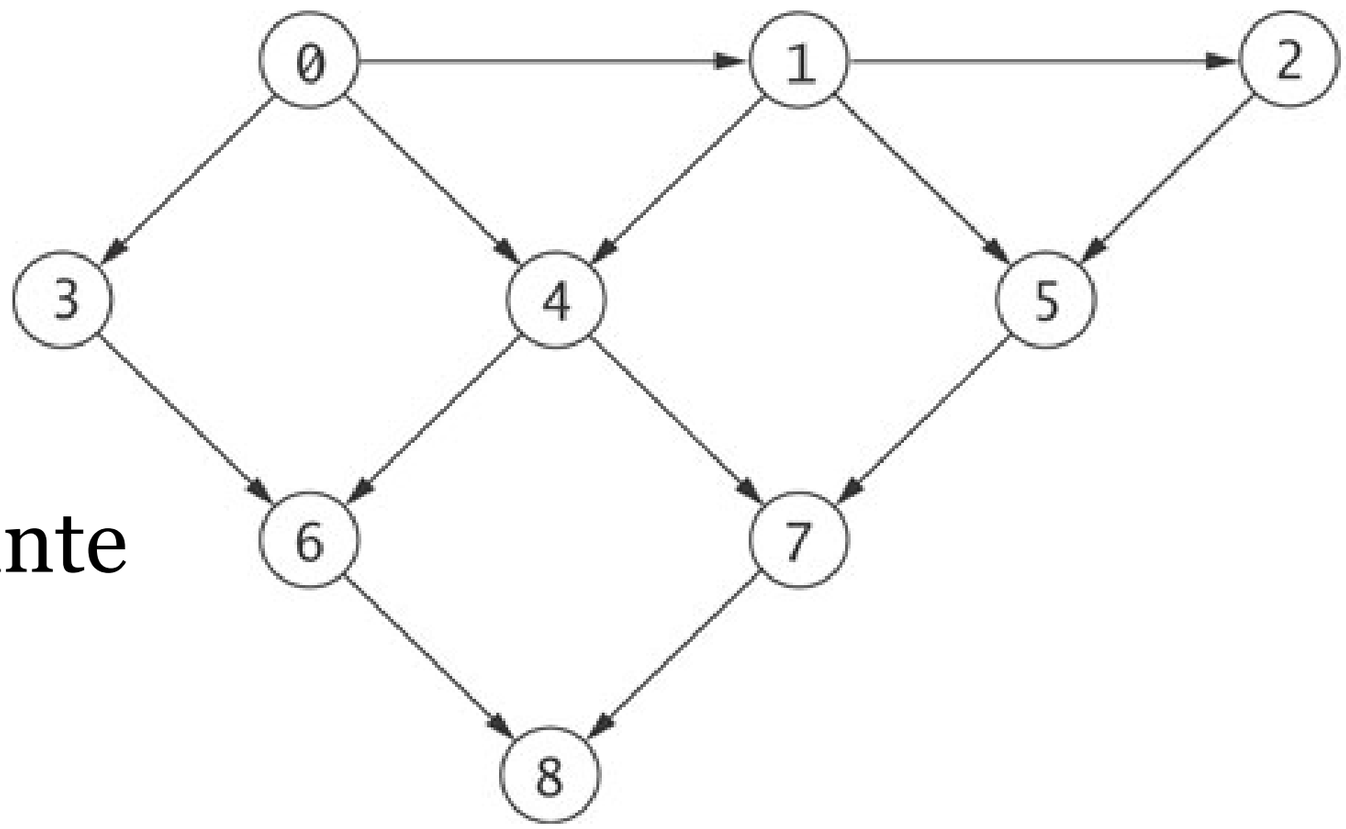
- en riktad graf utan cykler
- i en DAG kan man bara gå framåt, det finns ingen väg tillbaks



Exempel: Topologisk sortering

En topologisk sortering av noderna i en DAG, är att lista noderna i en sådan ordning att

- om (u, v) är en båge, så kommer u före v i listan
- varje DAG har minst en topologisk sortering, ofta fler än en
- 012345678 är en giltig topologisk sortering av denna DAG, men 015342678 är det inte



Exempel: Topologisk sortering

Antag att vi gör en djupet-först-sökning av en DAG:

- om det finns en båge (u, v) i grafen,
- så måste u bli klar efter att v är klar
- dvs, u måste komma efter v i avslutningsordningen

En enkel algoritm för topologisk sortering blir alltså:

- gör en djupet-först-sökning av grafen
- lista noderna i omvänd avslutningsordning

Summary

Graphs:

- typically implemented using adjacency lists or adjacency matrix
- can be directed, undirected, weighted, unweighted
- cyclic, acylic (*directed acyclic graph* (DAG) very common type)
- paths, cycles, strongly connected components

Traversals:

- breadth-first search
- depth-first search + topological sorting