Grafer

Koffman & Wolfgang

- kapitel 10, avsnitt 1–3

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.

Terminologi

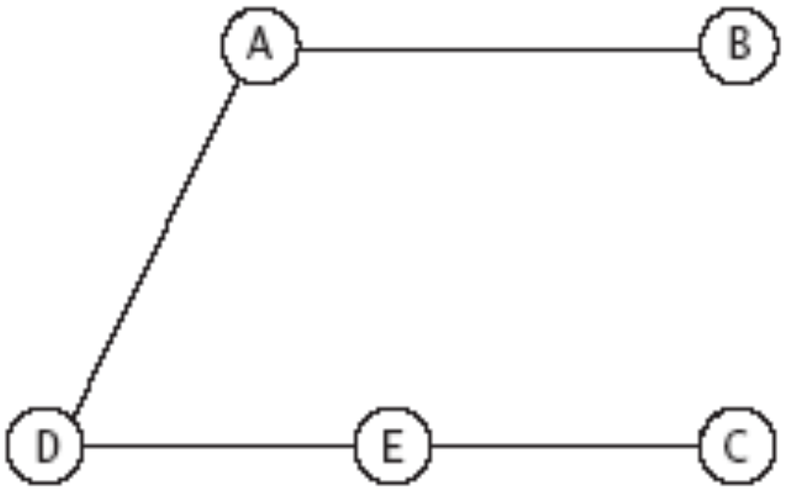
En graf är en datastruktur som består av en mängd *noder* (vertices) och en mängd *bågar* (edges)

- 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)

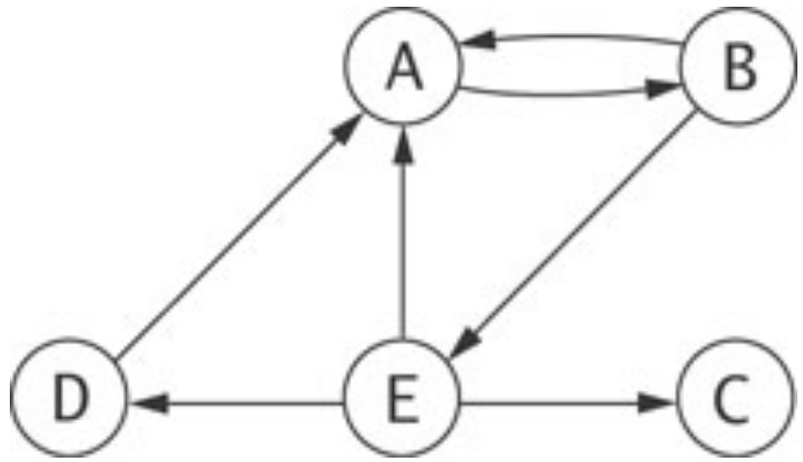
Visuell representation

Noder representeras som punkter eller cirklar

Bågar representeras som linjer mellan noderna



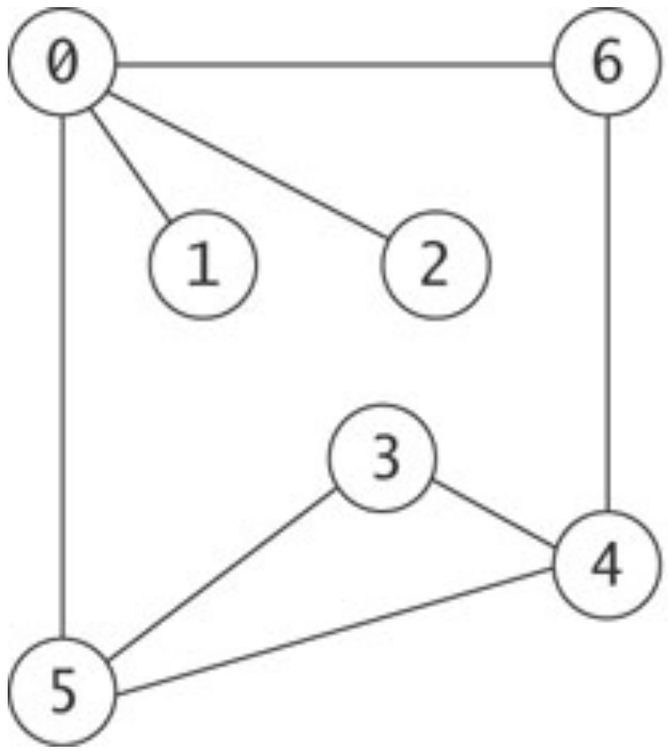
V = {A, B, C, D, E}
E = {(A, B), (A, D),
(C, E), (D, E)}

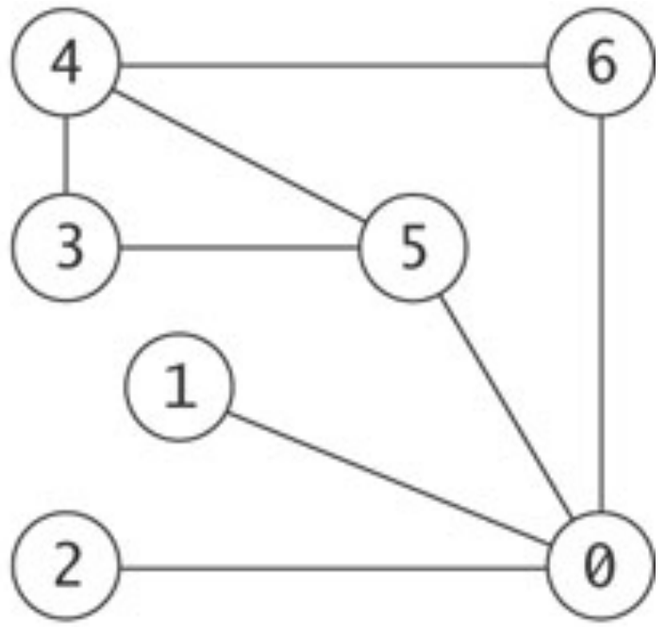


V = {A, B, C, D, E}
E = {(A, B), (B, A), (B, E), (D, A),
(E, A), (E, C), (E, D)}

Layouten är oviktig

Exakt hur vi placerar ut noderna och bågarna är oviktigt





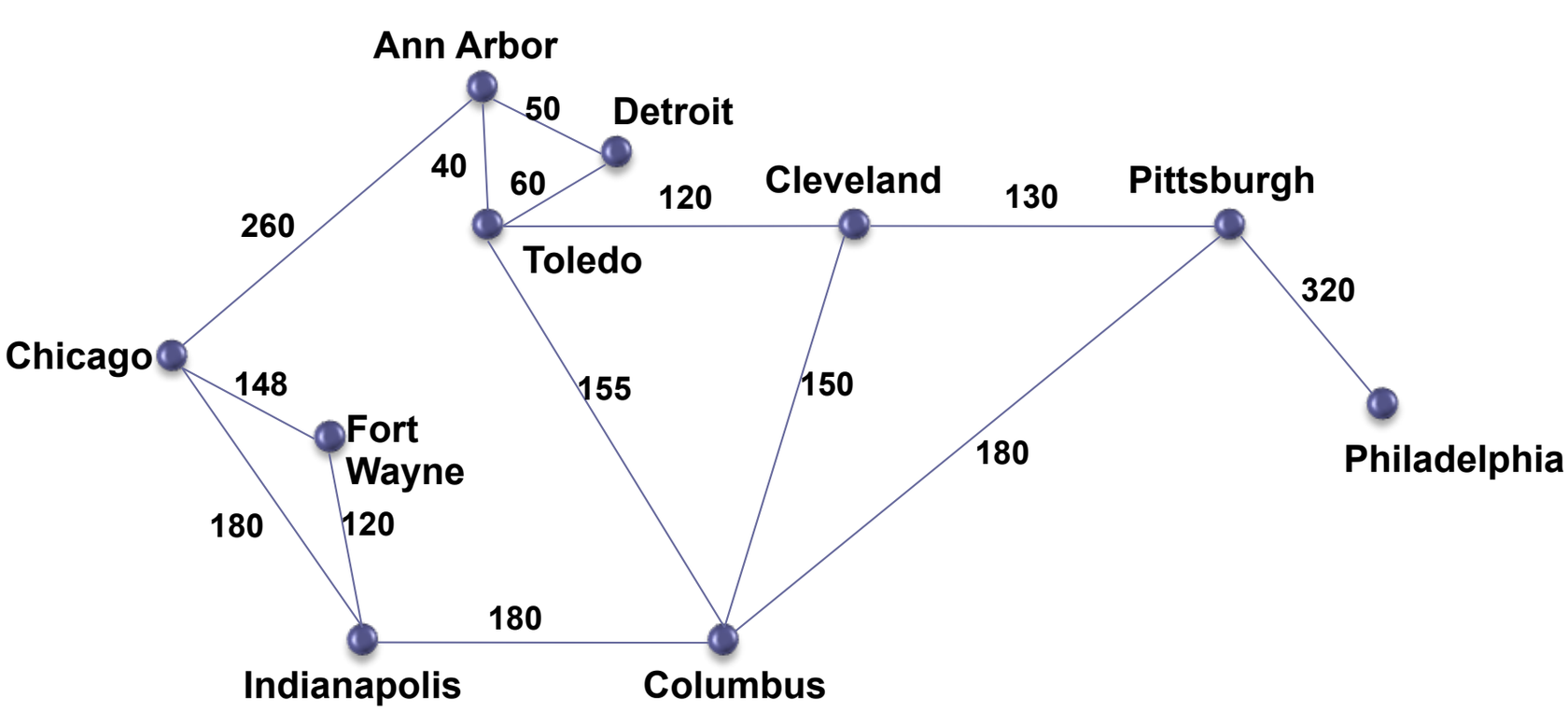
V = {0, 1, 2, 3, 4, 5, 6}
E = {(0, 1), (0, 2), (0, 5), (0, 6), (3, 5), (3, 4), (4, 5), (4, 6)}

Viktade grafer

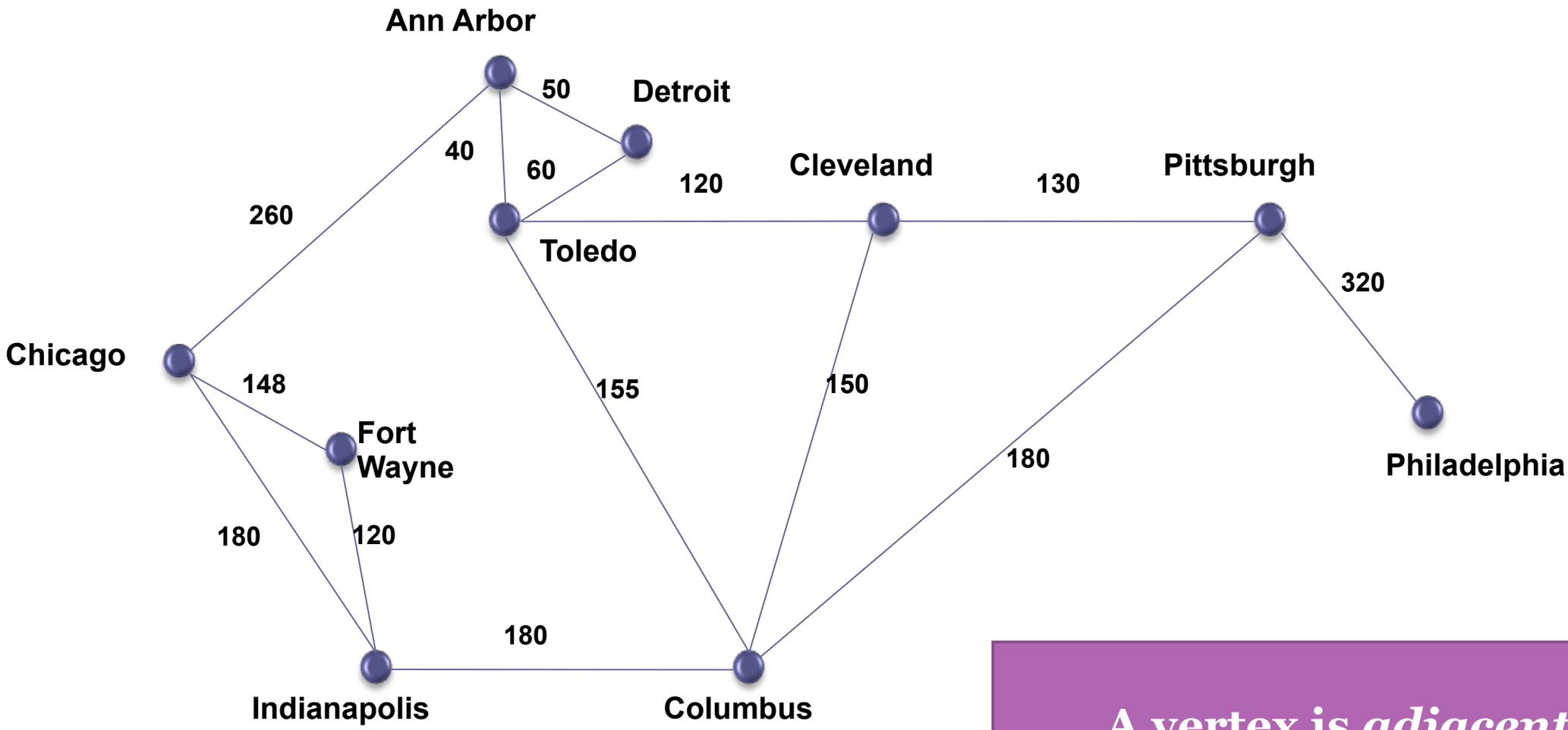
En båge kan ha en *vikt* kopplad till sig

- motsvarande graf blir då en *viktad* graf

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

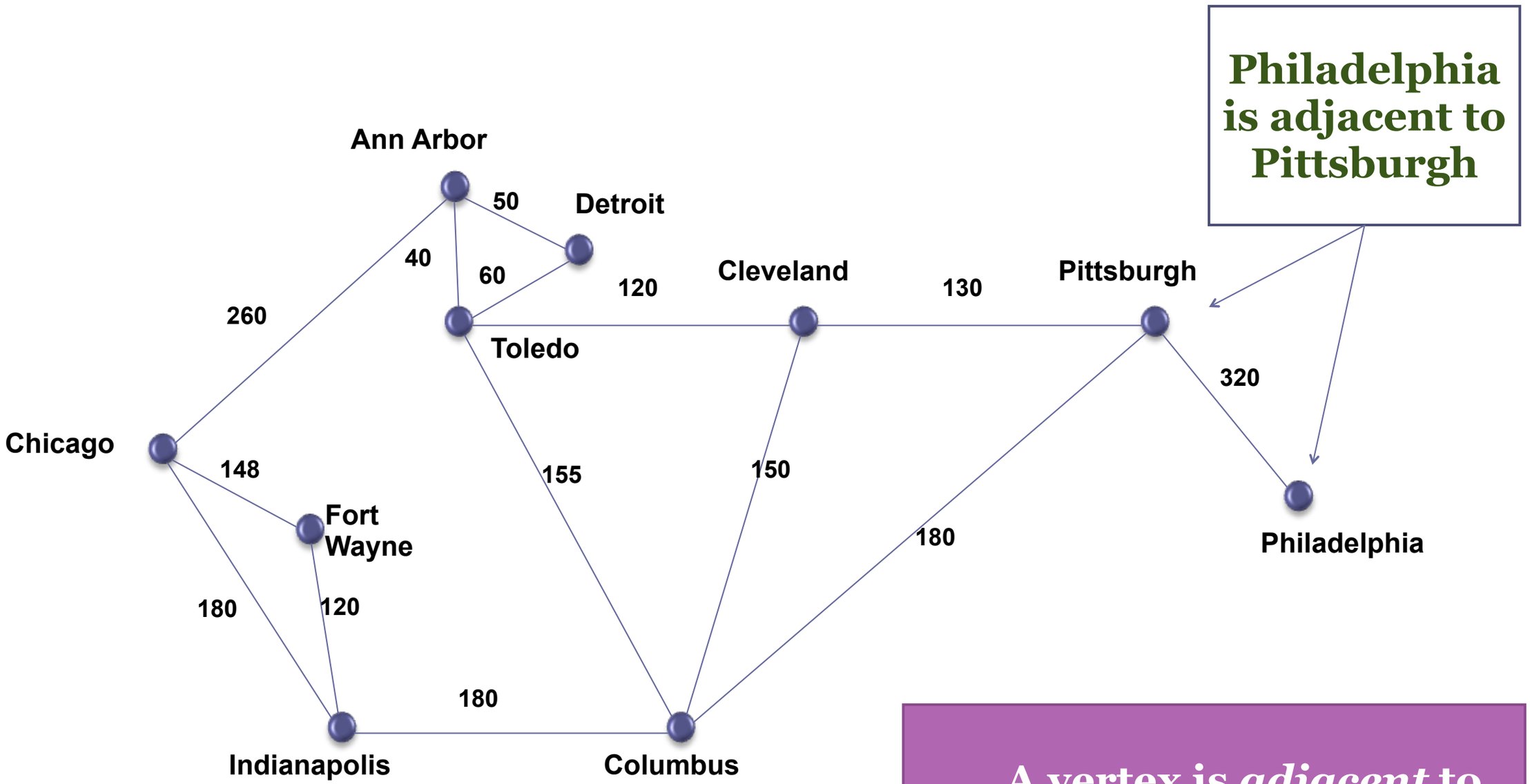


Stigar och cykler



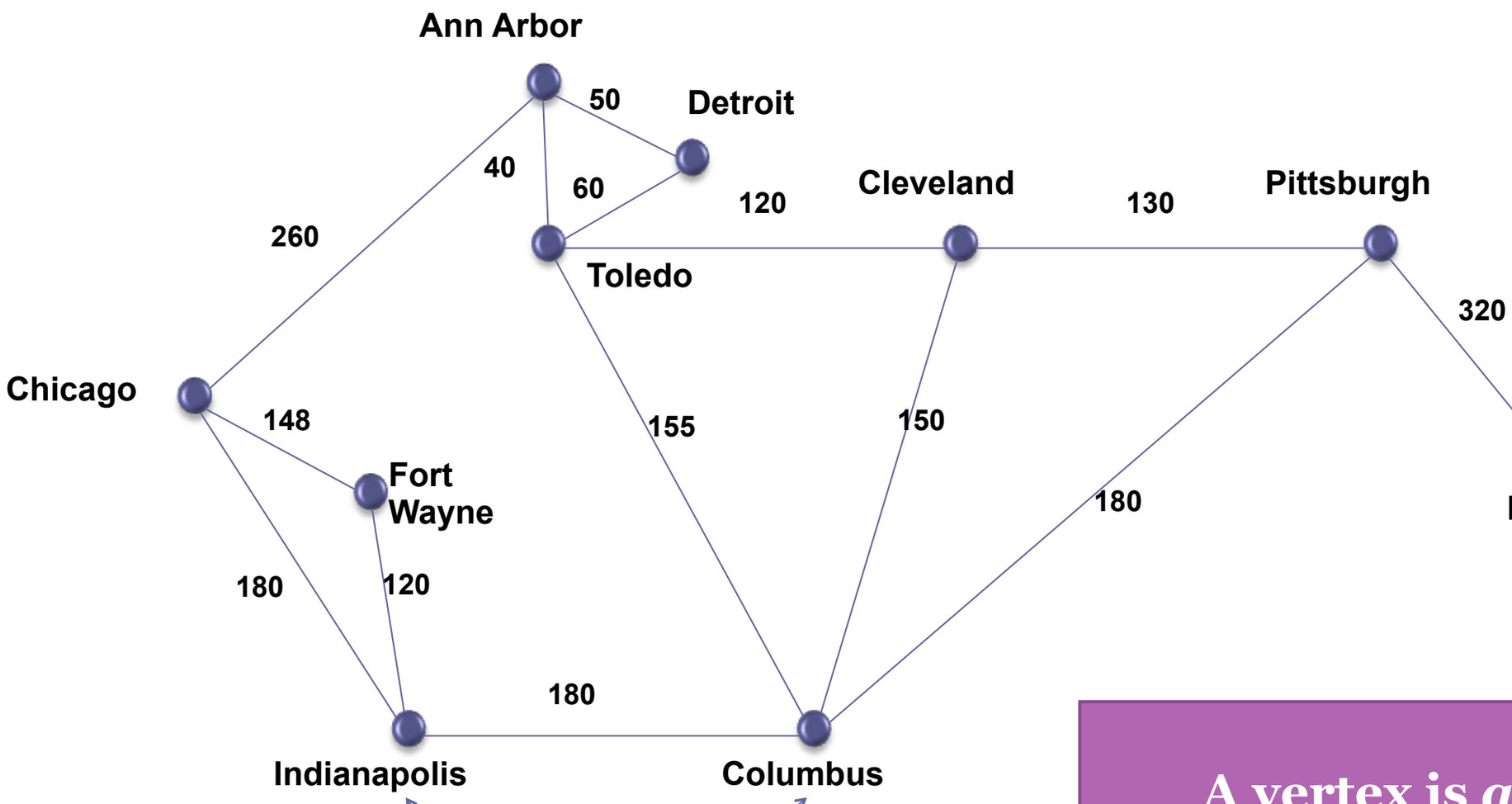
A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

Stigar och cykler



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

Stigar och cykler

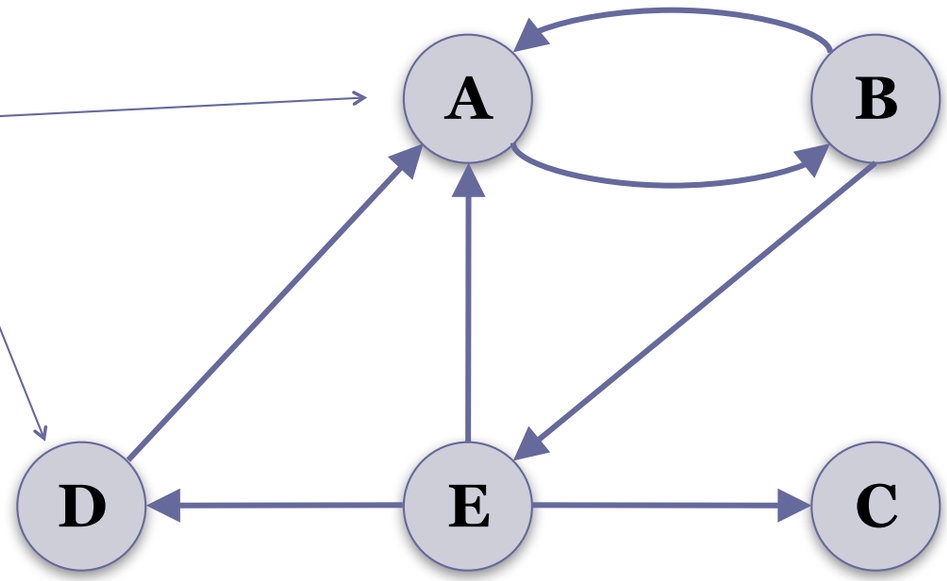


Indianapolis is adjacent to Columbus

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

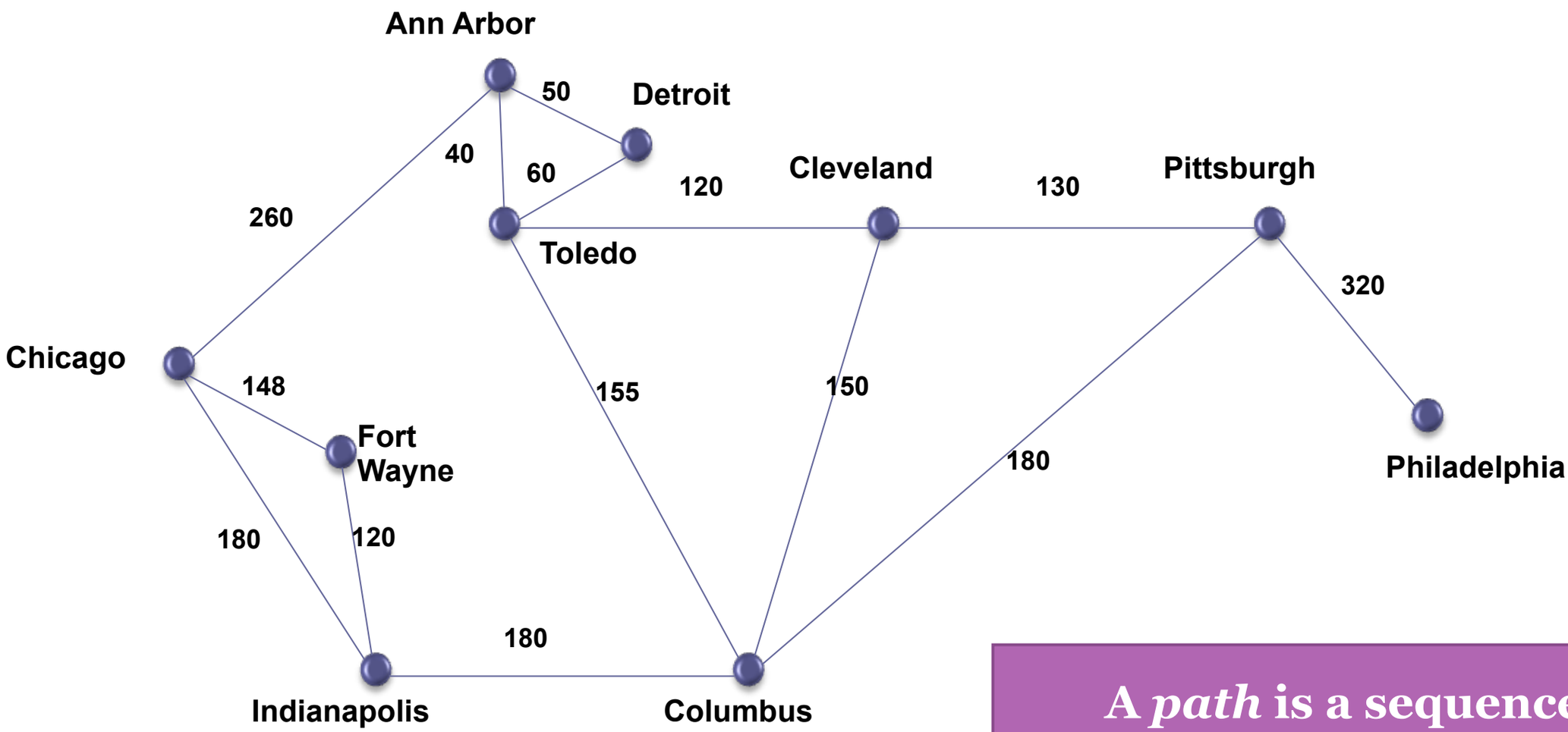
Stigar och cykler

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



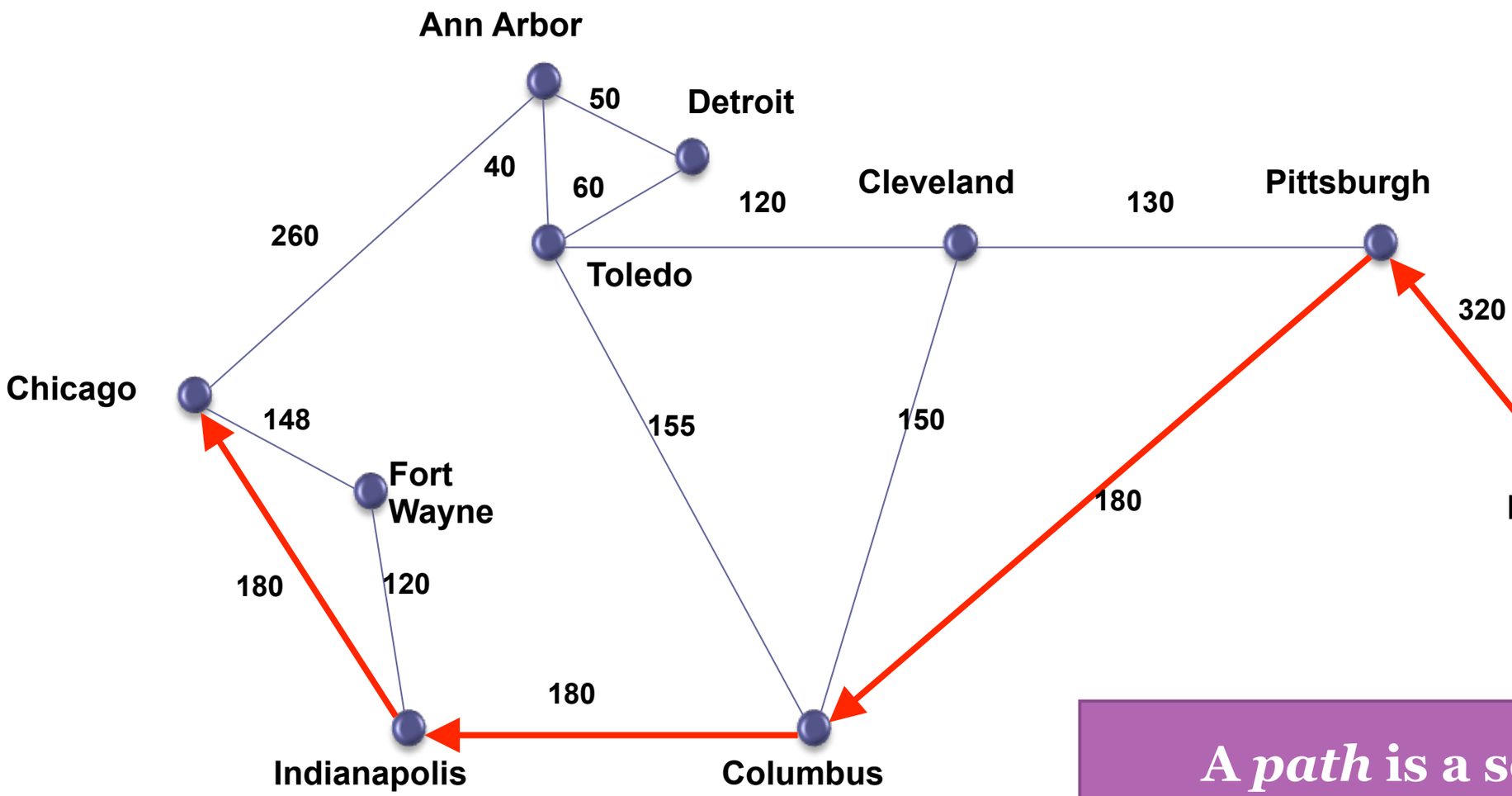
A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

Stigar och cykler



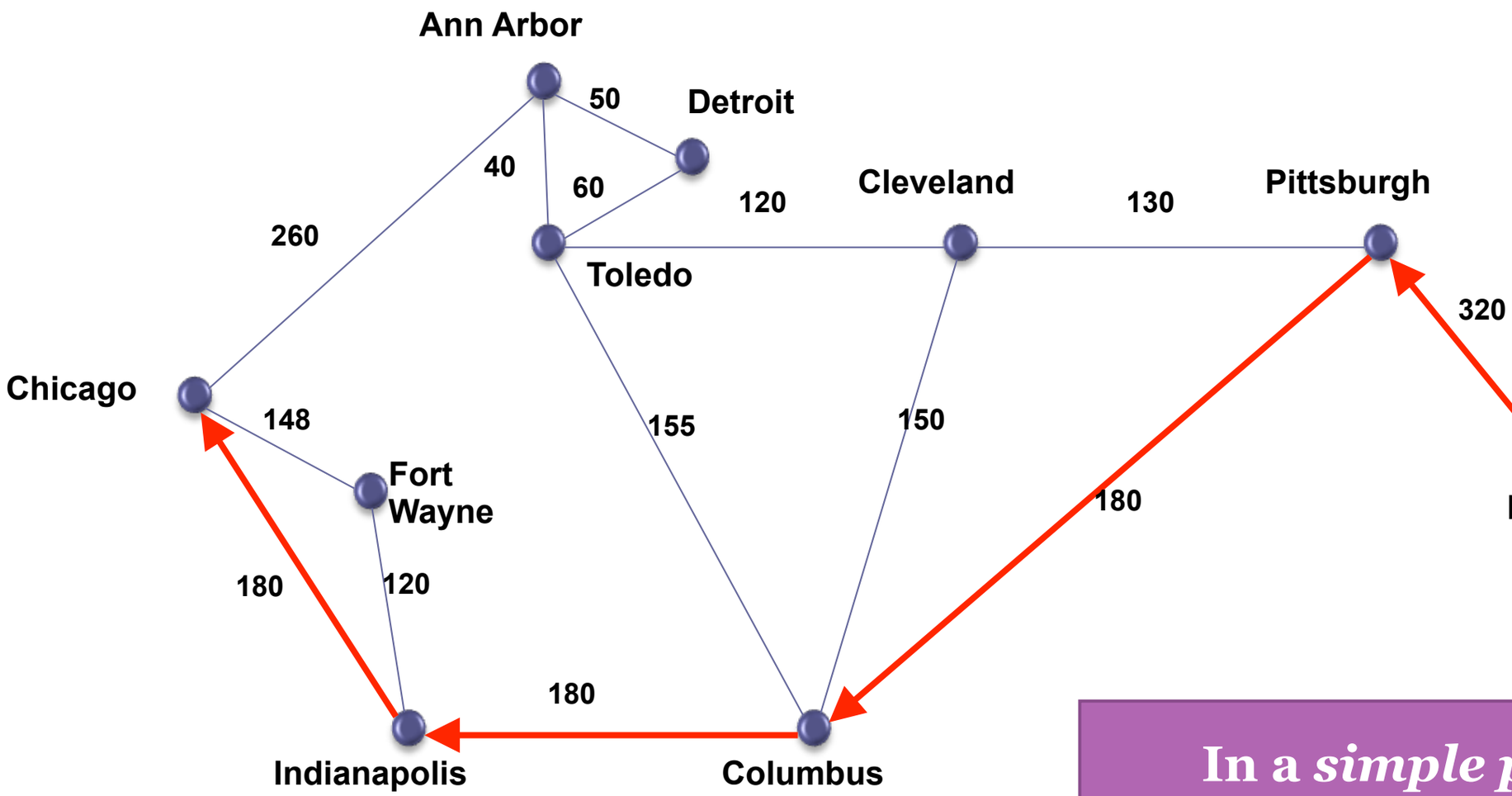
A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor

Stigar och cykler



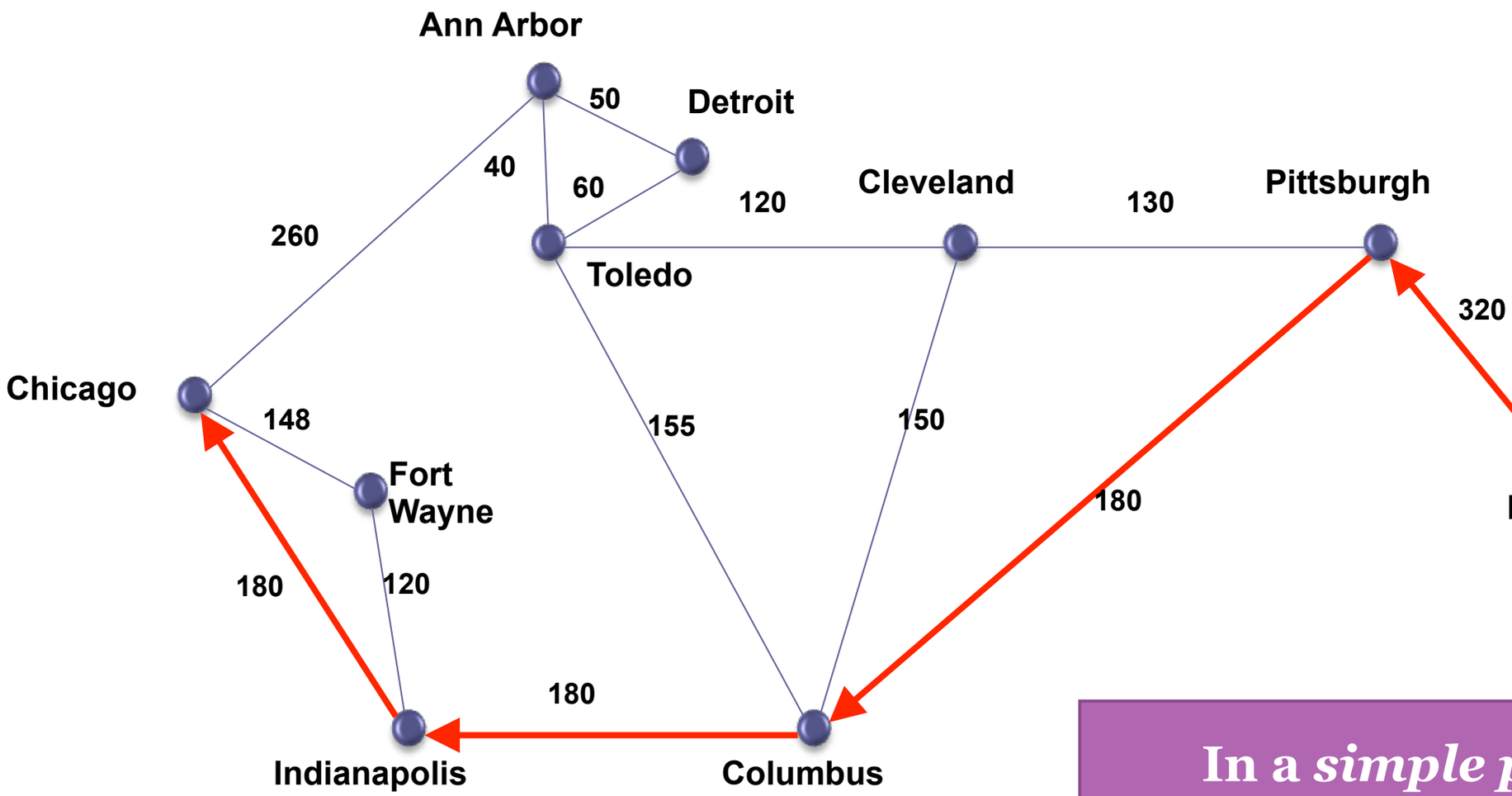
A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor

Stigar och cykler



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

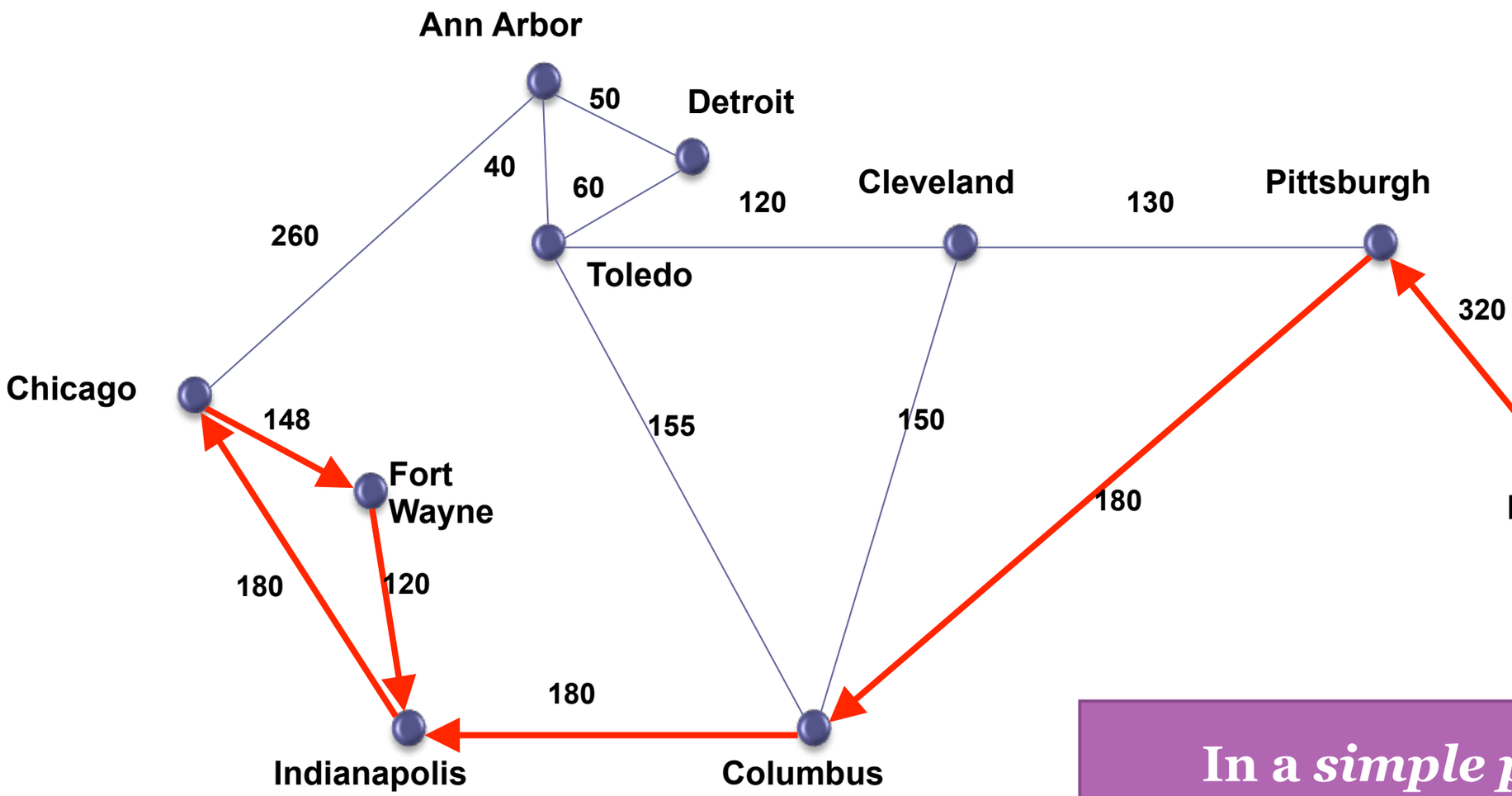
Stigar och cykler



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

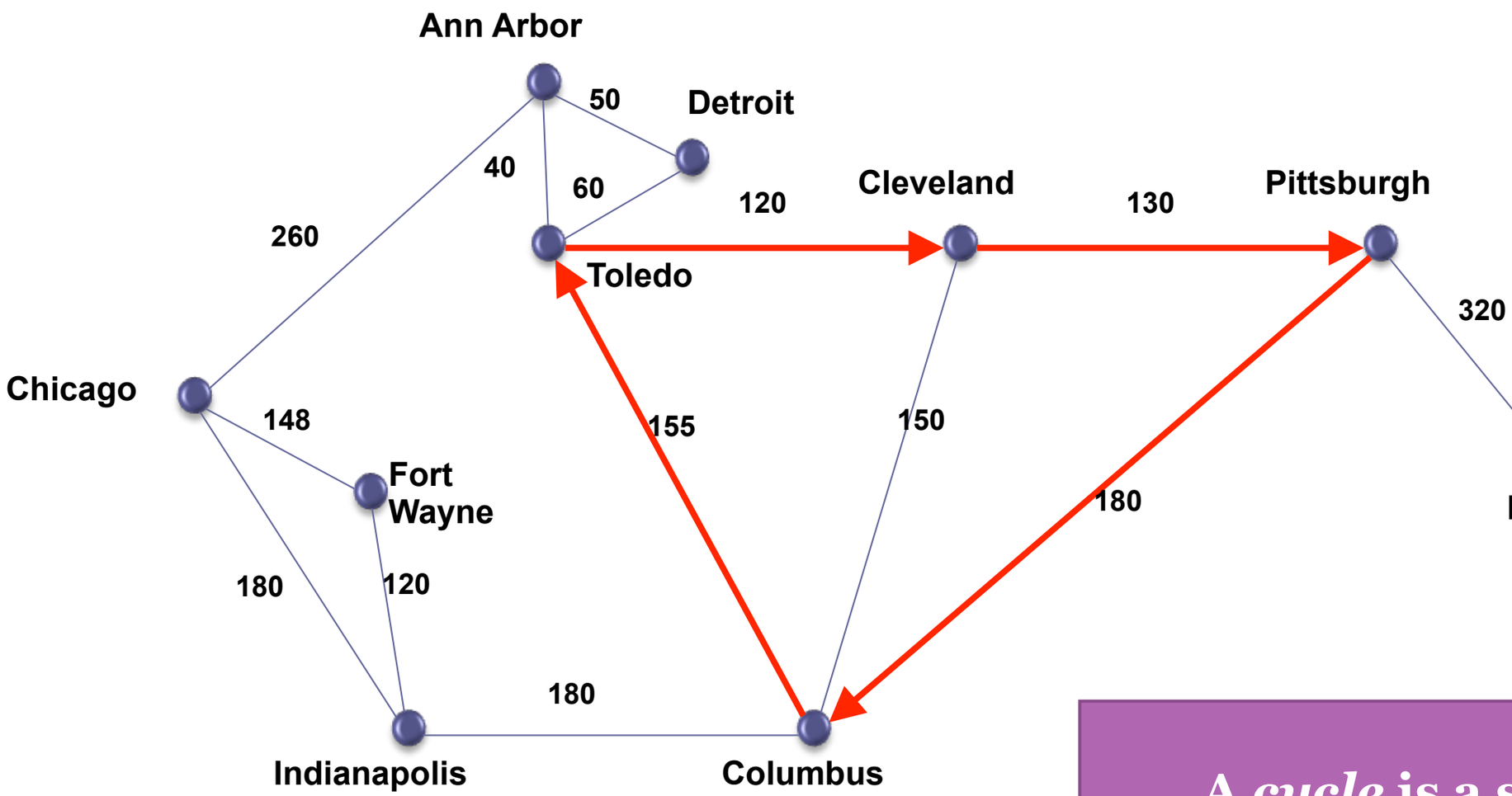
Stigar och cykler



This path is NOT a simple path

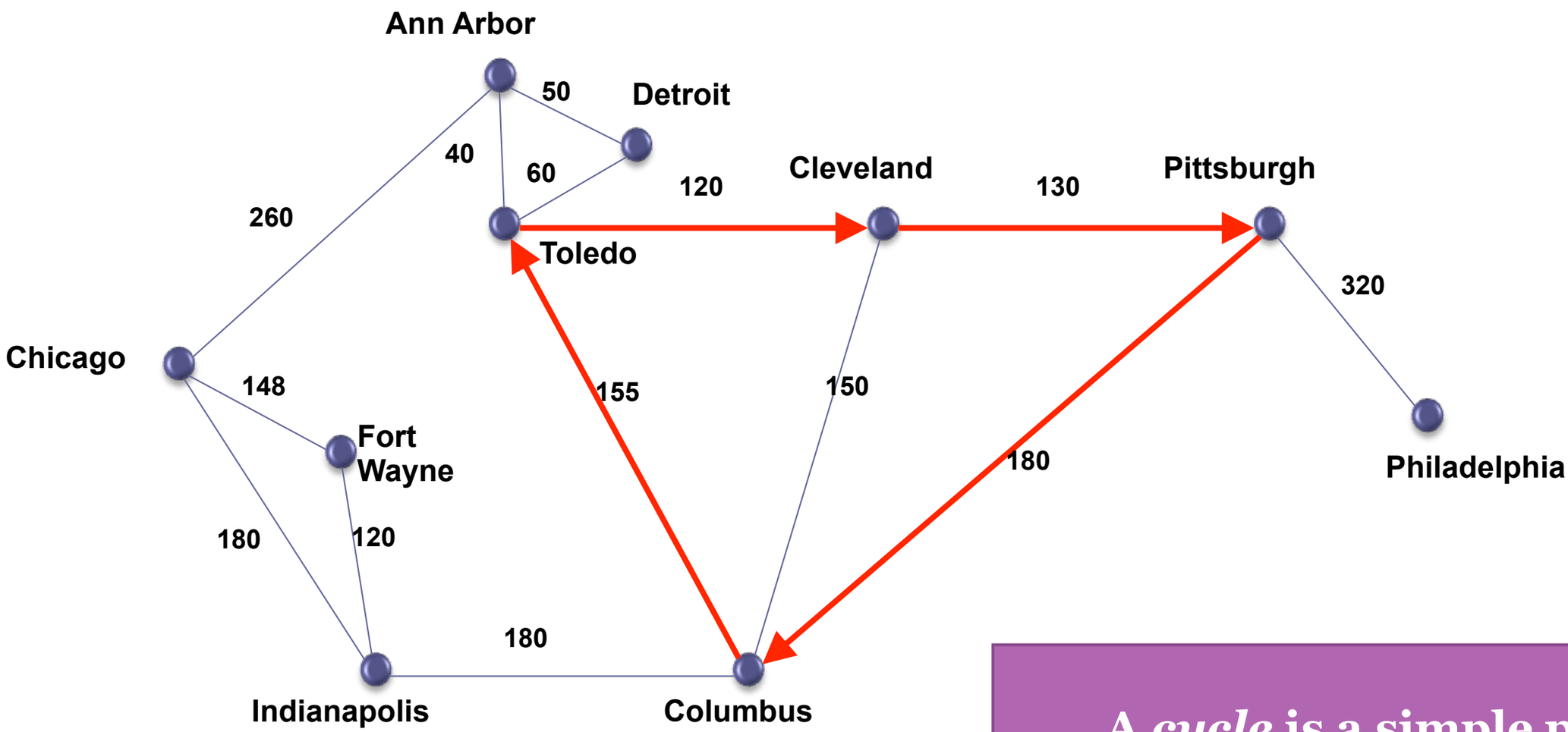
In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

Stigar och cykler



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

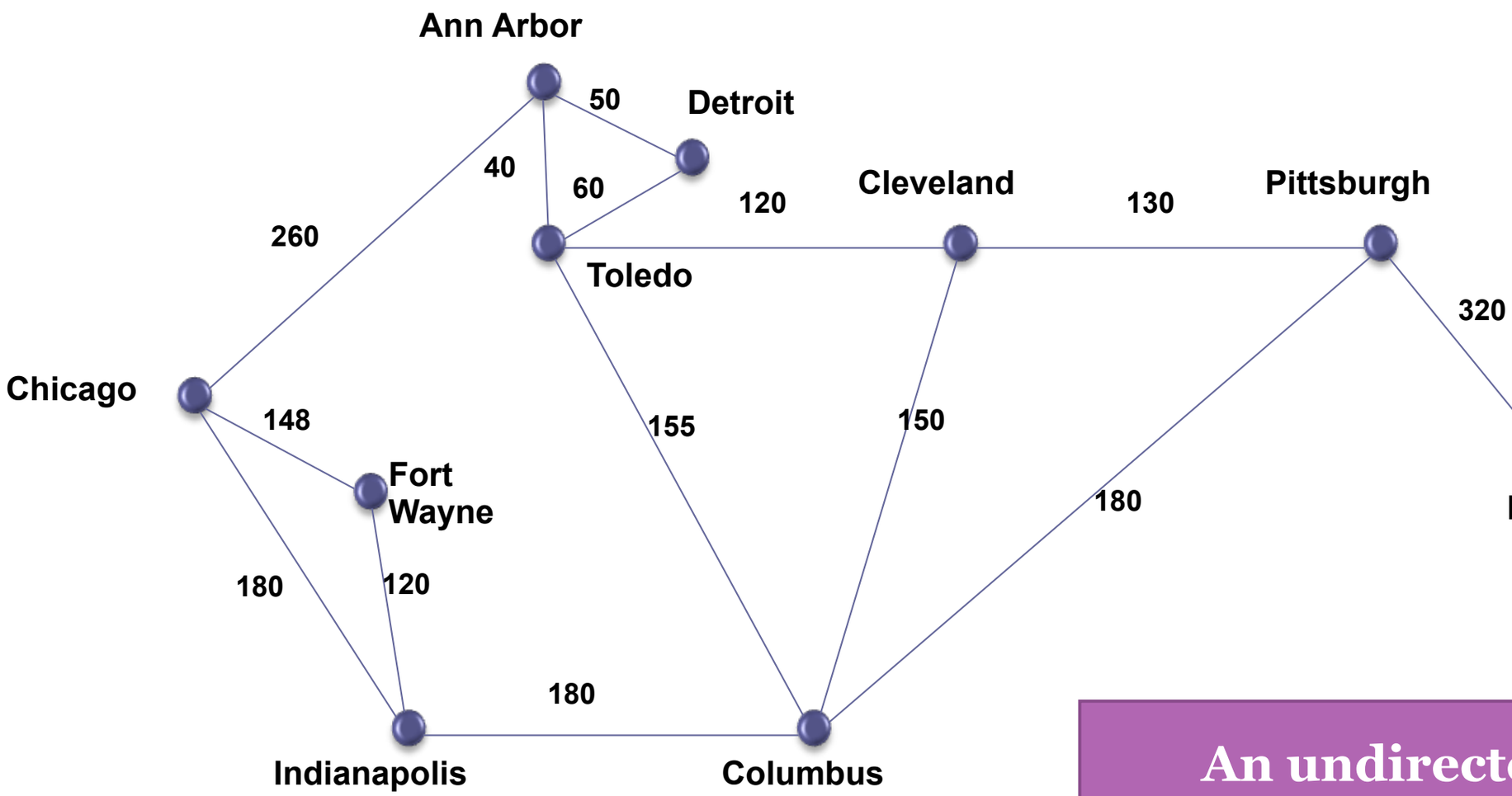
Stigar och cykler



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

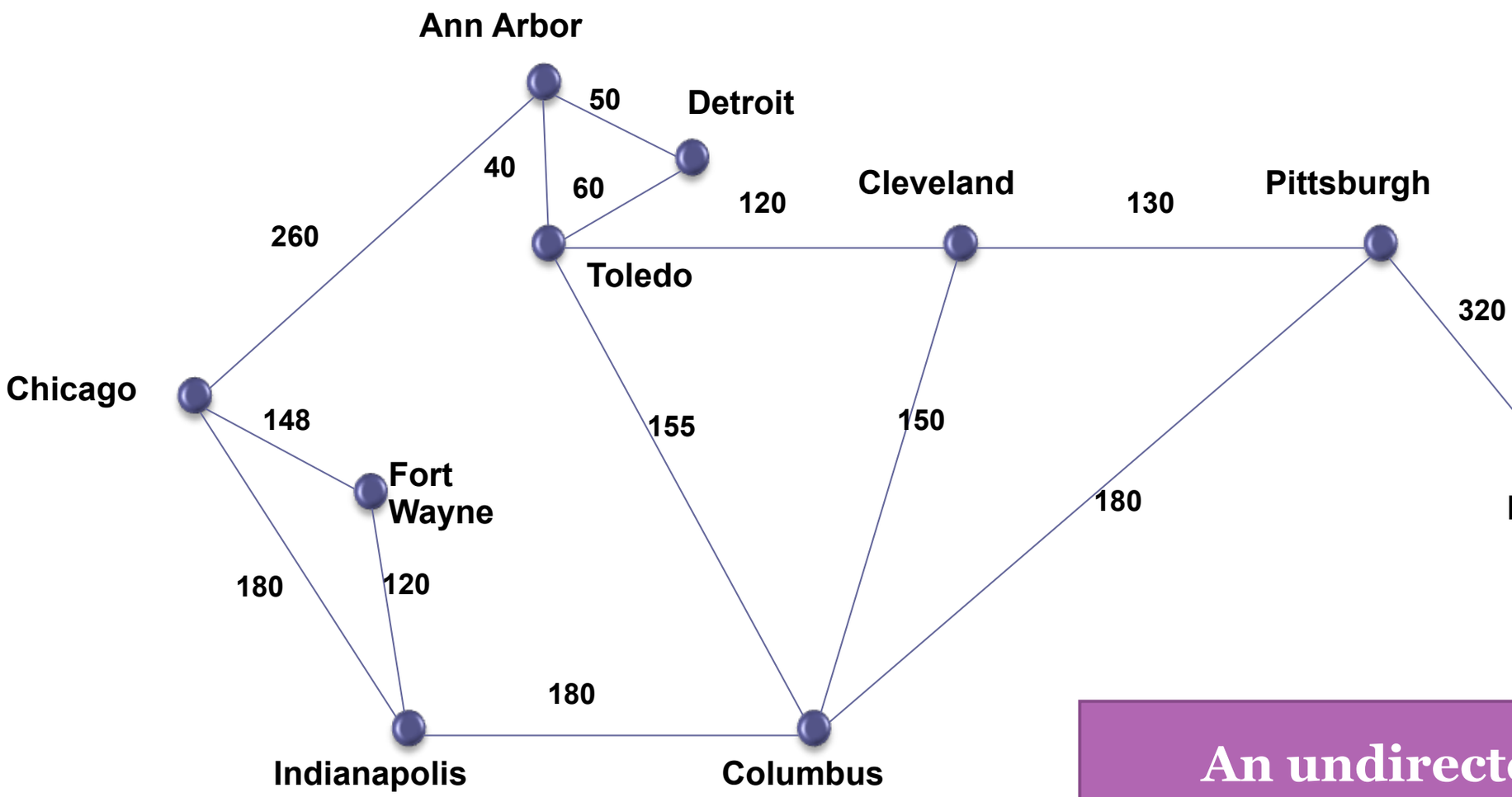
A *cycle* is a simple path in which only the first and final vertices are the same

Stigar och cykler



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

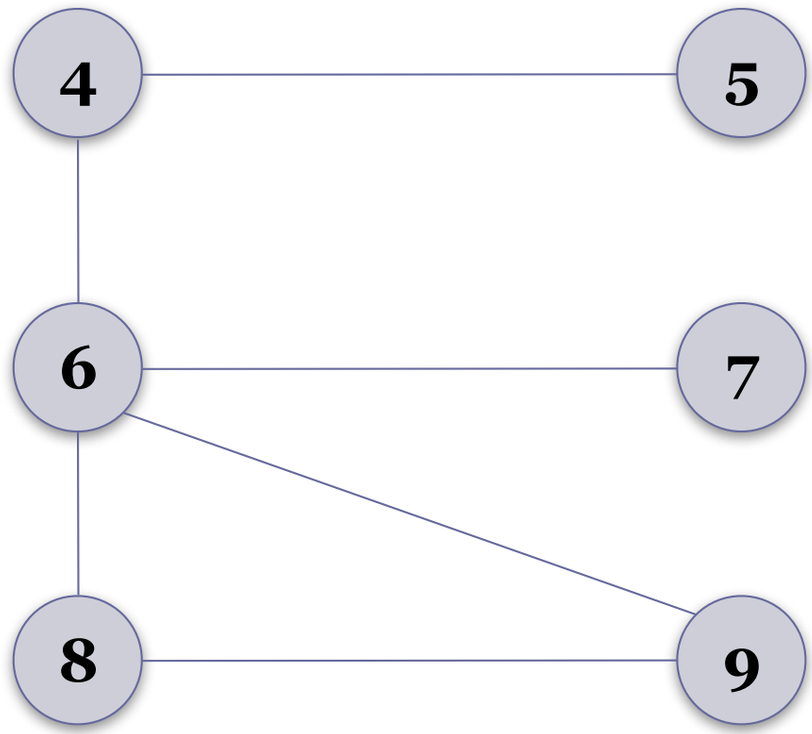
Stigar och cykler



This graph is a connected graph

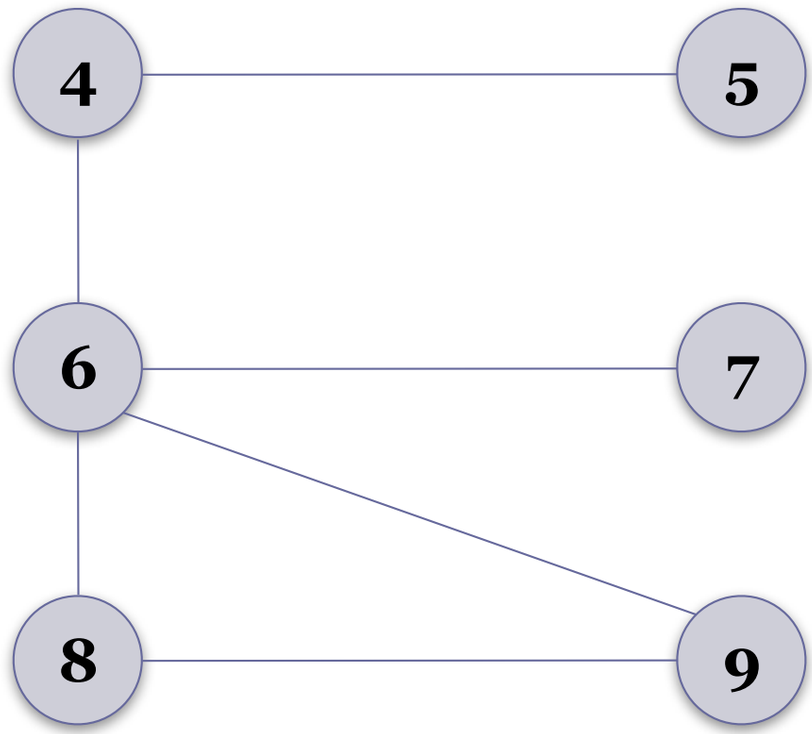
An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

Stigar och cykler



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

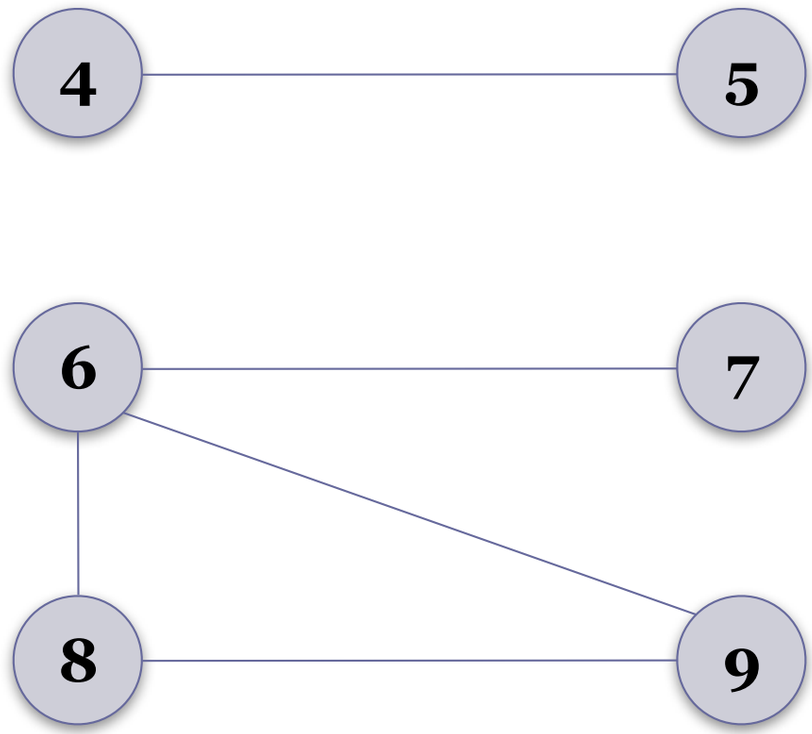
Stigar och cykler



This graph is a connected graph

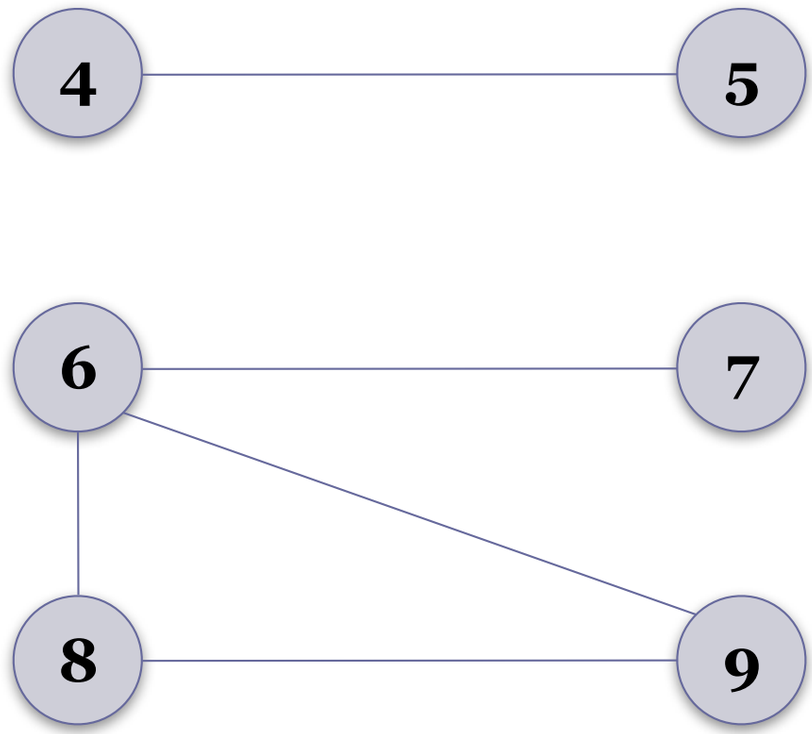
An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

Stigar och cykler



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

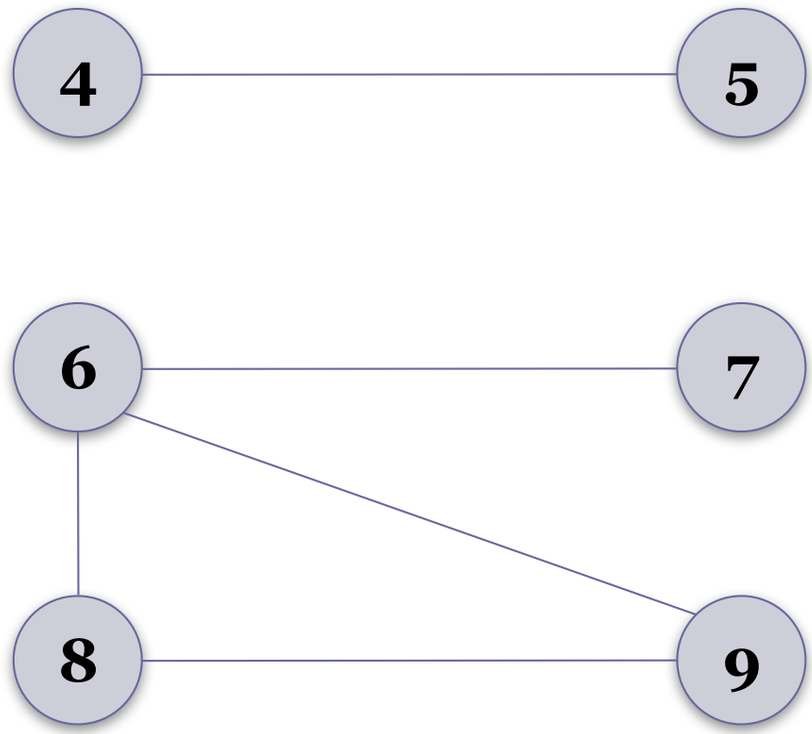
Stigar och cykler



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

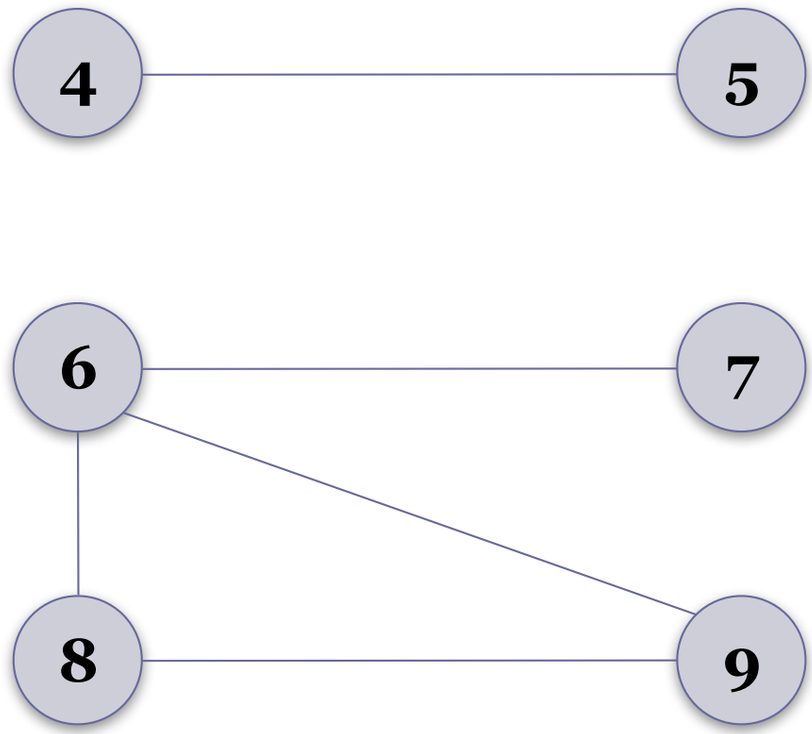
Stigar och cykler



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

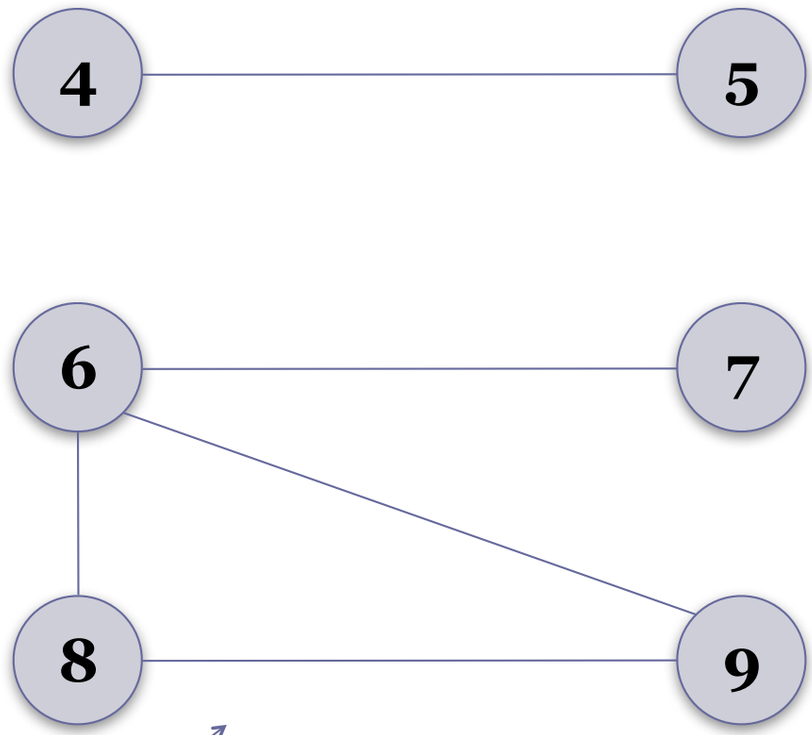
Stigar och cykler

{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

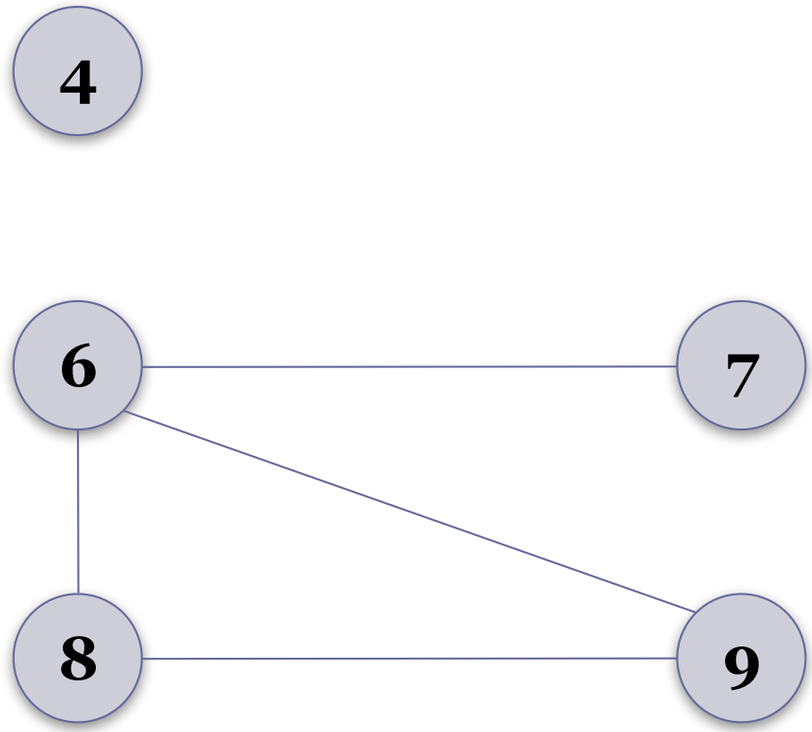


{6, 7, 8, 9} are connected components

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



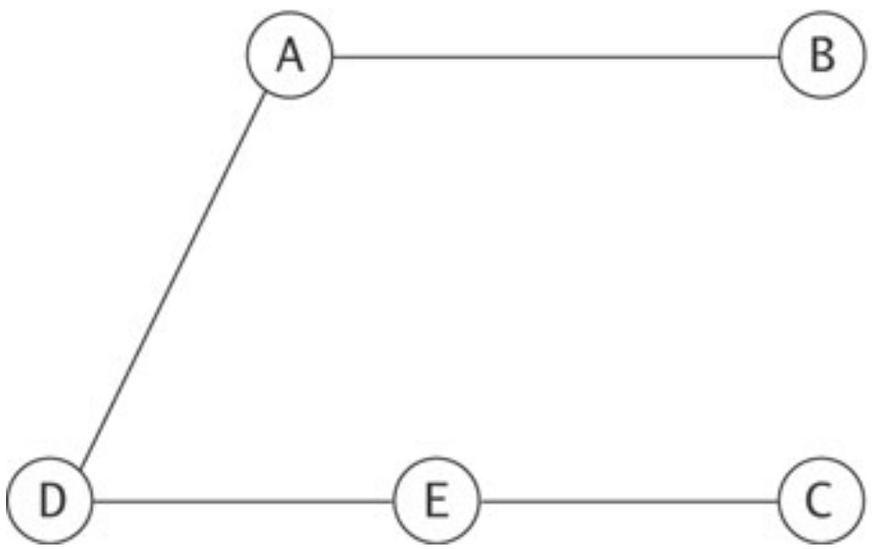
If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

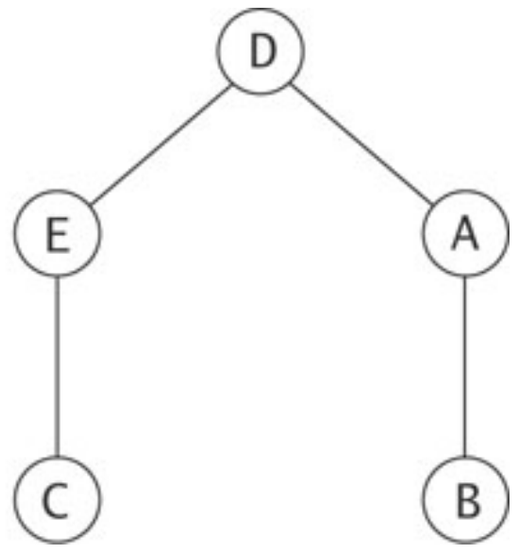
Träd är grafer

Träd är ett specialfall av grafer — varje graf som:

- är sammanhängande (connected), och
- inte innehåller några cykler

kan ses som ett träd genom att göra någon av noderna till rot





Grafer i Java

Java har inget inbyggt graf-gränssnitt.

Kursboken ger ett eget, med följande metoder:

- tillverka en graf av en viss storlek
- lägga till en båge i grafen
- iterera genom alla noder i grafen
- iterera genom alla noder som är angränsande (adjacent) till en given nod
- avgöra om det finns en båge mellan två noder
- returnera vikten hos en båge mellan två noder

Noder och bågar i Java

Noder:

- vi representerar noder med heltal k, $0 \leq k < |V|$
- där $|V|$ betyder kardinaliteten, dvs antalet noder i V

Bågar:

- vi definierar klassen Edge som innehåller
  - en nod ’*source*’ (från-noden)
  - en nod ’*destination*’ (till-noden)
  - ett flyttal ’*weight*’ (vikten)
  - oviktade bågar använder vikten 1.0
- alla bågar är riktade
- oriktade grafer måste ha två Edge-objekt för varje båge (en Edge i varje riktning)

Klassen Edge (tabell 10.1)

Data Field	Attribute
`private int dest`	The destination vertex for an edge.
`private int source`	The source vertex for an edge.
`private double weight`	The weight.
Constructor	**Purpose**
`public Edge(int source, int dest)`	Constructs an `Edge` from `source` to `dest`. Sets the `weight` to 1.0.
`public Edge(int source, int dest, double w)`	Constructs an `Edge` from `source` to `dest`. Sets the `weight` to `w`.
Method	**Behavior**
`public boolean equals(Object o)`	Compares two edges for equality. Edges are equal if their source and destination vertices are the same. The weight is not considered.
`public int getDest()`	Returns the destination vertex.
`public int getSource()`	Returns the source vertex.
`public double getWeight()`	Returns the weight.
`public int hashCode()`	Returns the hash code for an edge. The hash code depends only on the source and destination.
`public String toString()`	Returns a string representation of the edge.

Att implementera grafer

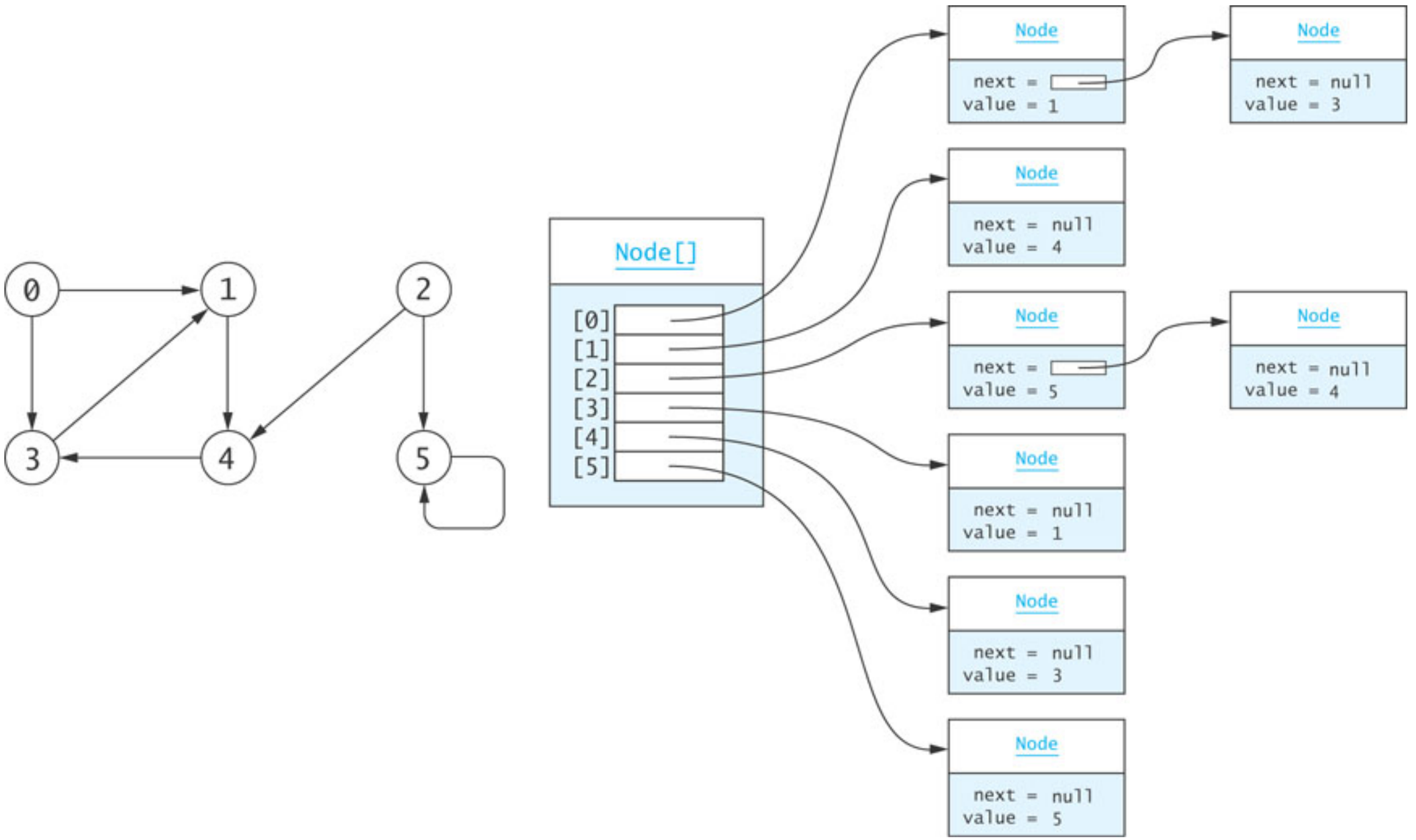
Alternativ 1: Bågarna representeras av en array med |V| listor (kallad ”adjacency lists”), en lista för varje nod

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

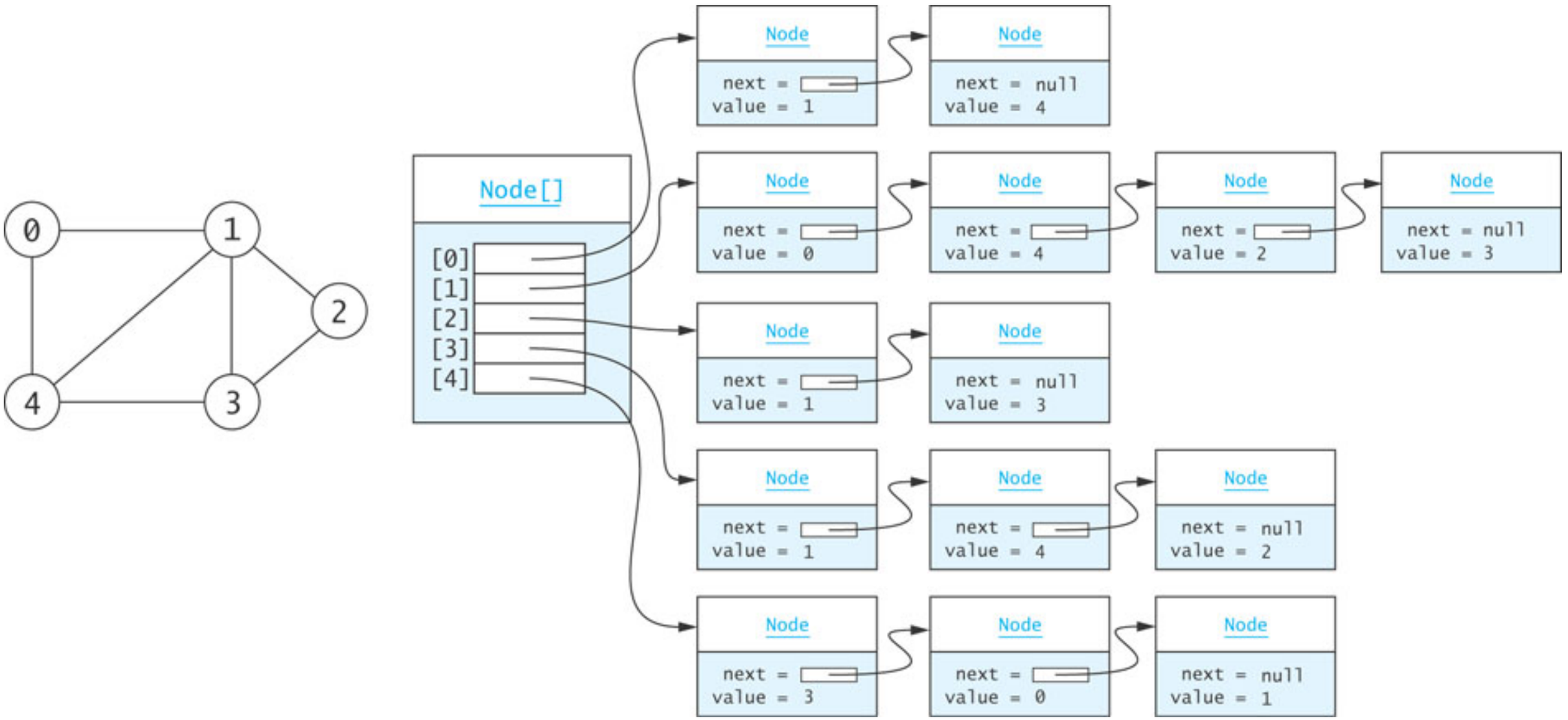
Alternativ 2: Bågarna representeras av en 2-dimensionell array (kallad ”adjacency matrix”), med |V| rader och |V| kolumner

- 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

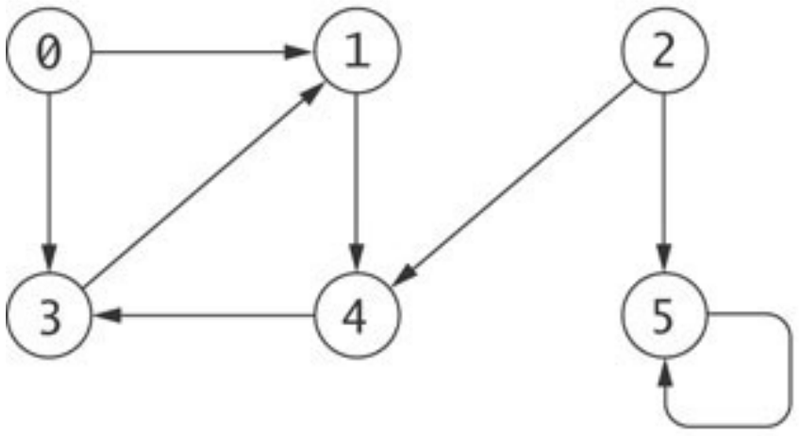
För en viktad graf, kommer cellerna att innhålla vikterna:

- 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:

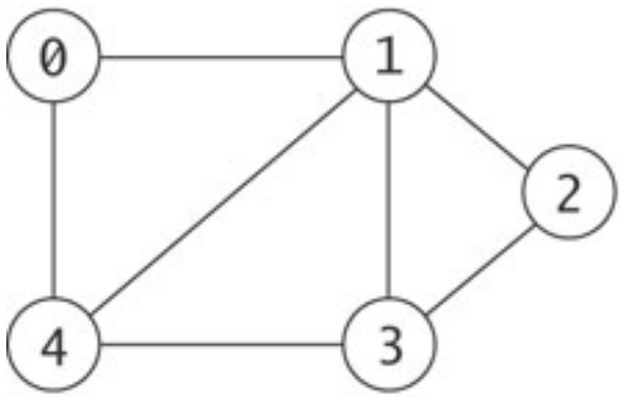
- vi behöver alltså bara använda den nedre vänstra triangeln

Adjacency matrix, oviktad



Column

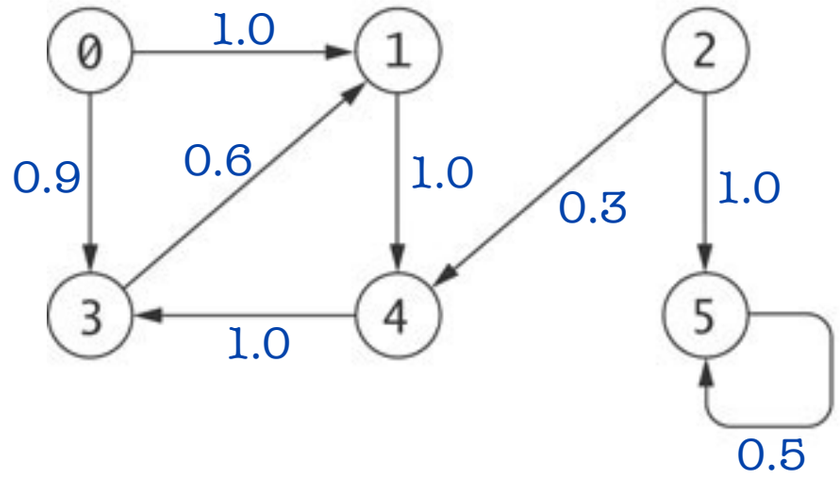
Row	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		1.0		
[1]					1.0	
[2]					1.0	1.0
[3]		1.0				
[4]				1.0		
[5]						1.0



Column

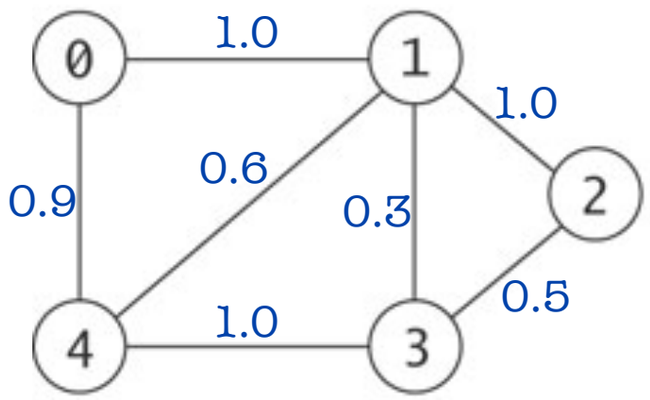
Row	[0]	[1]	[2]	[3]	[4]
[0]		1.0			1.0
[1]	1.0		1.0	1.0	1.0
[2]		1.0		1.0	
[3]		1.0	1.0		1.0
[4]	1.0	1.0		1.0	

Adjacency matrix, viktad



Column

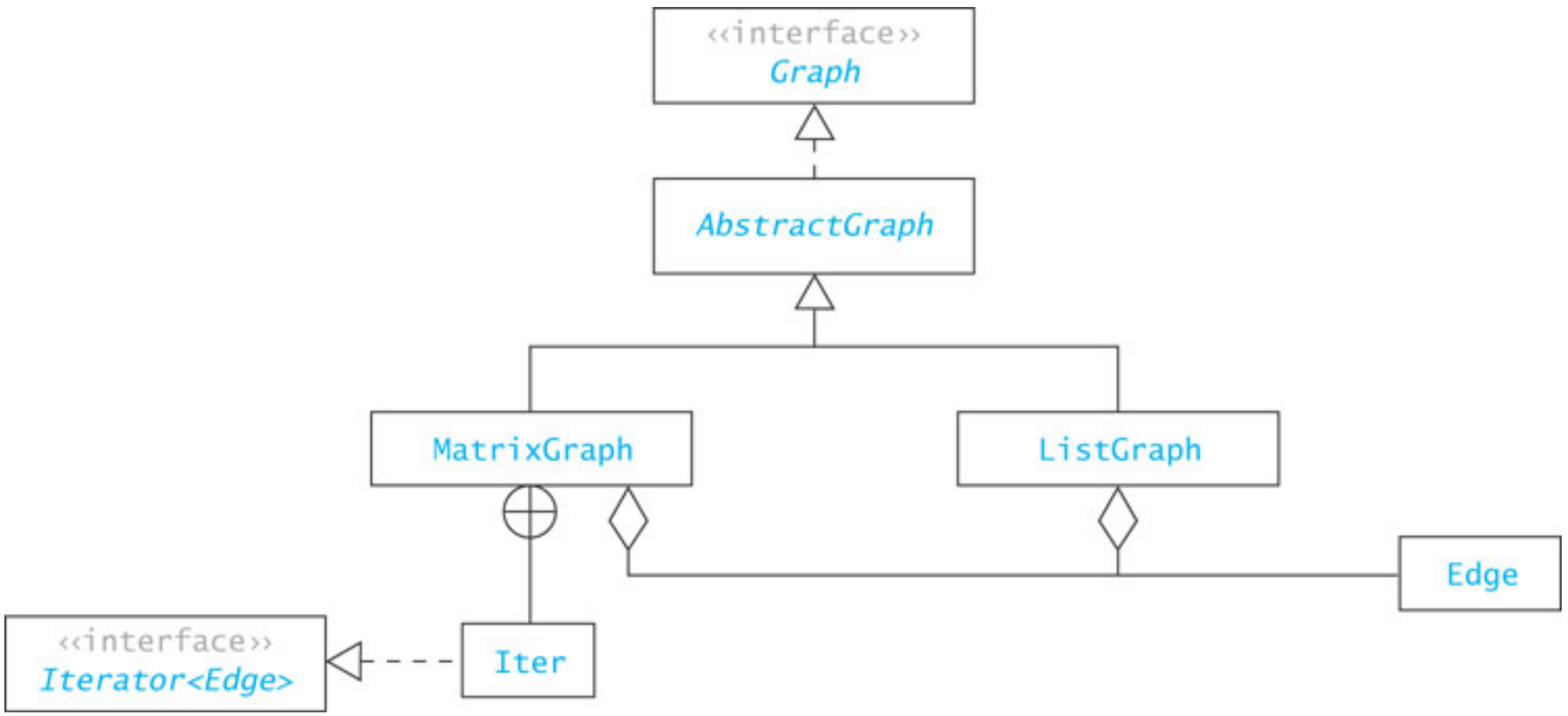
Row	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		0.9		
[1]					1.0	
[2]					0.3	1.0
[3]		0.6				
[4]				1.0		
[5]						0.5



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	

Översikt över graf-hierarkin



Klasserna AbstractGraph, ListGraph och MatrixGraph

Några metoder är lika för list- och matris-representationen, t.ex.:

```
int  getNumV()          // antalet noder i grafen
boolean  isDirected()  // om grafen är riktad
```

- därför finns superklassen AbstractGraph

Klassen ListGraph har en instansvariabel:

```
private  List<Edge>[] edges
```

Klassen MatrixGraph har istället:

```
private  double[][] edges
```

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