Graph Implementations 1

CS 225 - Fall 2025

Mattox Beckman

Objectives

Objectives

- Implement a graph using an Adjacency Matrix
- Implement a graph using an Adjacency List
- Compare these to the Edge List implementation

Graph ADT

Modification

- `insertVertex(K key)`
- `insertEdge(Vertex v1, Vertex v2, K Key)`
- `removeVertex(K key)`
- `removeEdge(Vertex v1, Vertex v2)`

Need storage for vertices and keys. Maybe weights.

Graph ADT

Query

- `getEdges(Vertex v)`
- `areAdjecent(Vertex v1,Vertex v2)`
- `origin(Edge e)`
- `destination(Edge e)`

How to pick?

Three common ways to implement graphs

- Edge List (vector of edges)
- Adjacency List (vector of lists)
- Adjacency Matrix (2d vector)

How do you pick?

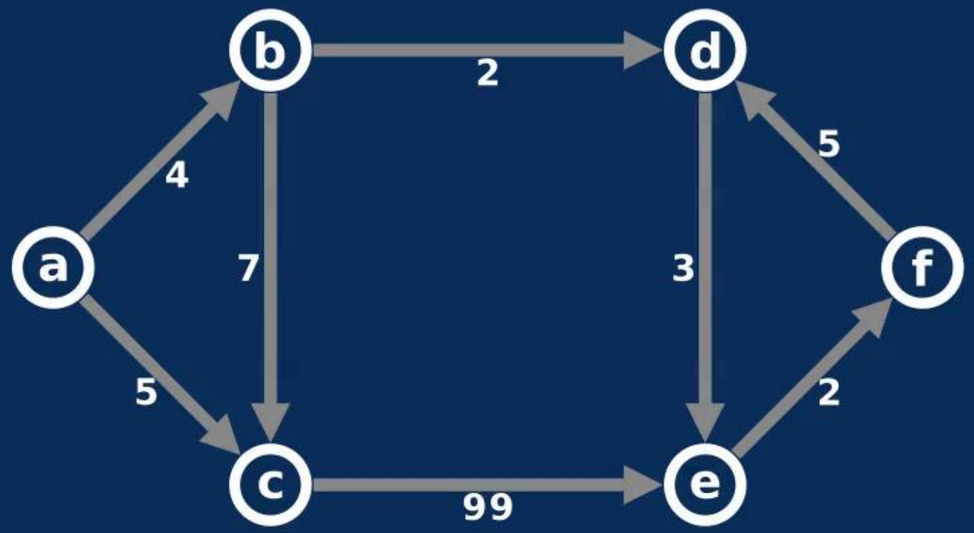
Last time we showed Edge List

Data Representation Affects the Questions You Can Ask

Example

Edge List

a	b	4
a	c	5
b	c	7
b	d	2
c	e	99
d	e	3
e	f	2
f	d	5



a	b	c	d	e	f

Vertex List

Timing 1

Edge List

a	b	4
a	c	5
b	c	7
b	d	2
c	e	99
d	e	3
e	f	2
f	d	5

How long do things take?

`insertVertex(K key)`

· $\mathcal{O}(1)$

`insertEdge(Vertex v1, Vertex v2, K Key)`

· $\mathcal{O}(m)$ if you avoid duplicates, $\mathcal{O}(1)$ otherwise

`removeVertex(K key)`

· $\mathcal{O}(m)$ (need to remove edges)

`removeEdge(Vertex v1, Vertex v2)`

· $\mathcal{O}(m)$ (need to remove edges)

a	b	c	d	e	f

Vertex List

Timing, 2

Edge List

a	b	4
a	c	5
b	c	7
b	d	2
c	e	99
d	e	3
e	f	2
f	d	5

How long do things take?

`getEdges(Vertex v)`

- $\mathcal{O}(m)$ - Check both start and end!

`areAdjacent(Vertex v1, Vertex v2)`

- $\mathcal{O}(m)$

`origin(Edge e)`

- $\mathcal{O}(m)$ or $\mathcal{O}(1)$

a	b	c	d	e	f

Vertex List

Tradeoffs

Pros

- Compact representation
- Edge iteration is easy
- Good for Minimum Spanning Trees

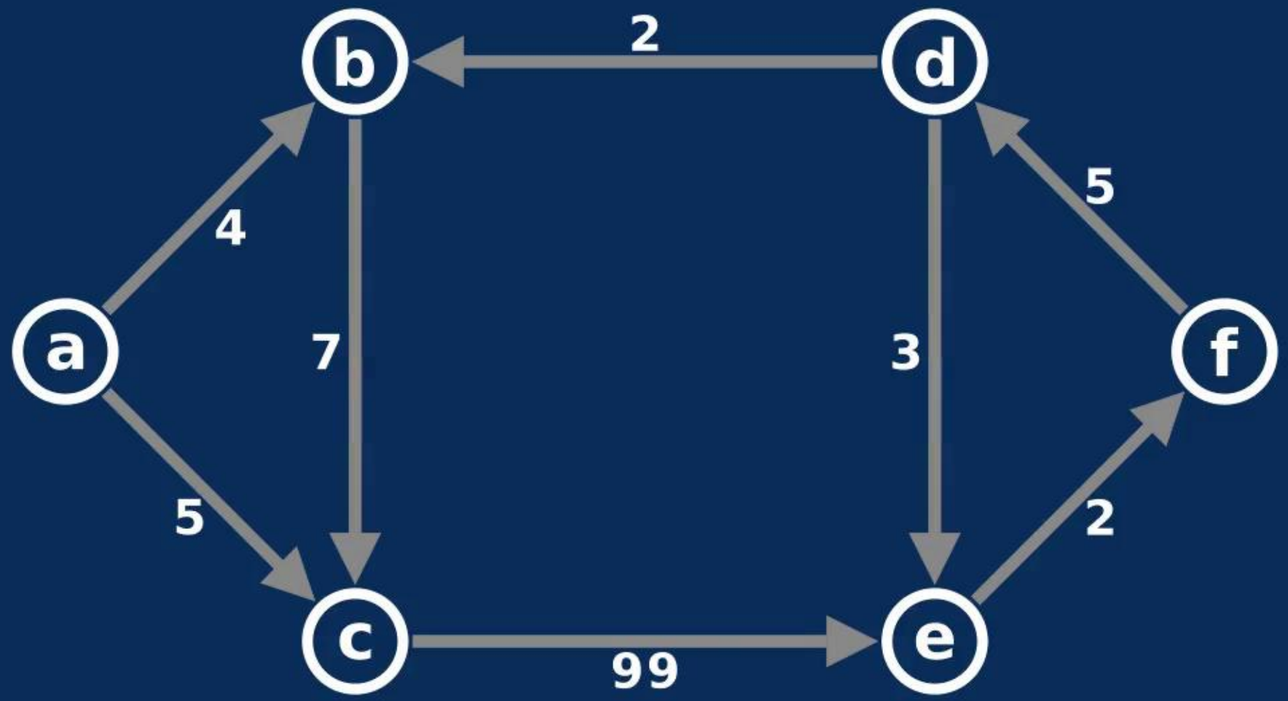
Cons

- DFS and BFS are slow!
- Vertex oriented things are slow

Adjacency Matrix

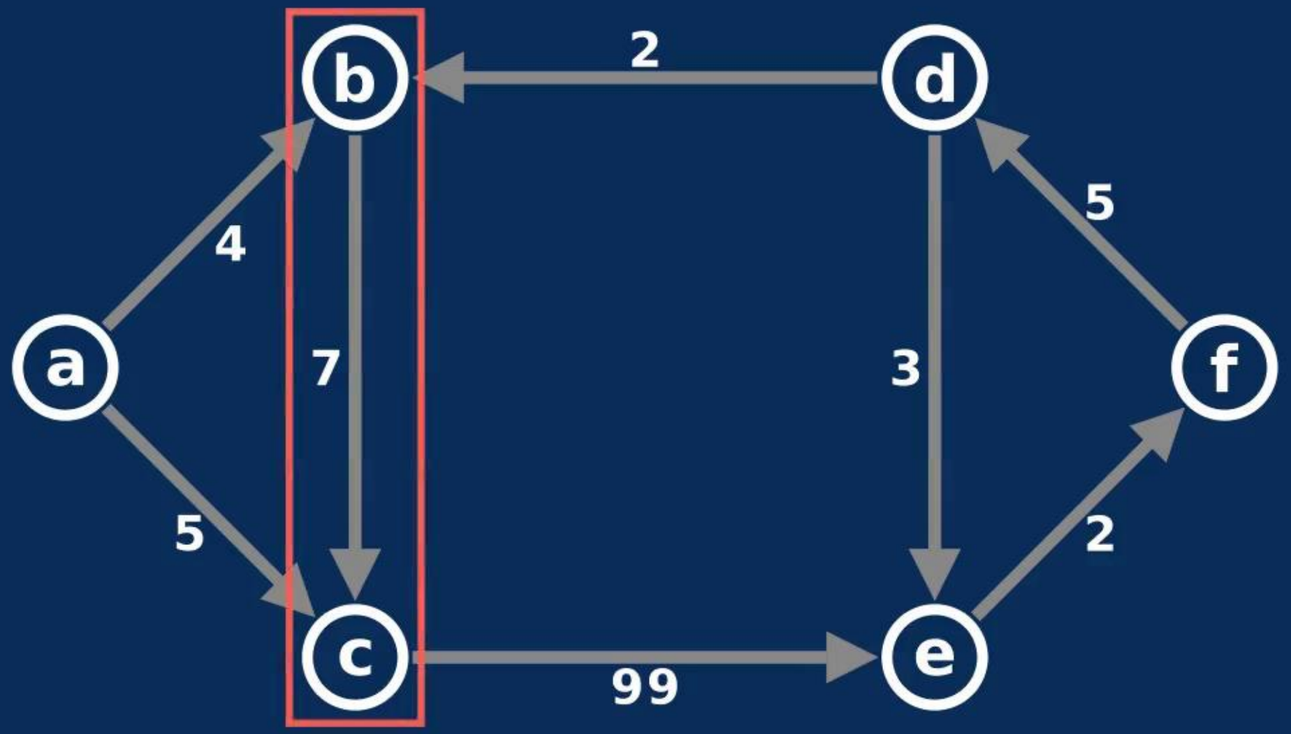
Introduction

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	0	0	0	0	0	0
c	0	0	0	0	0	0
d	0	0	0	0	0	0
e	0	0	0	0	0	0
f	0	0	0	0	0	0



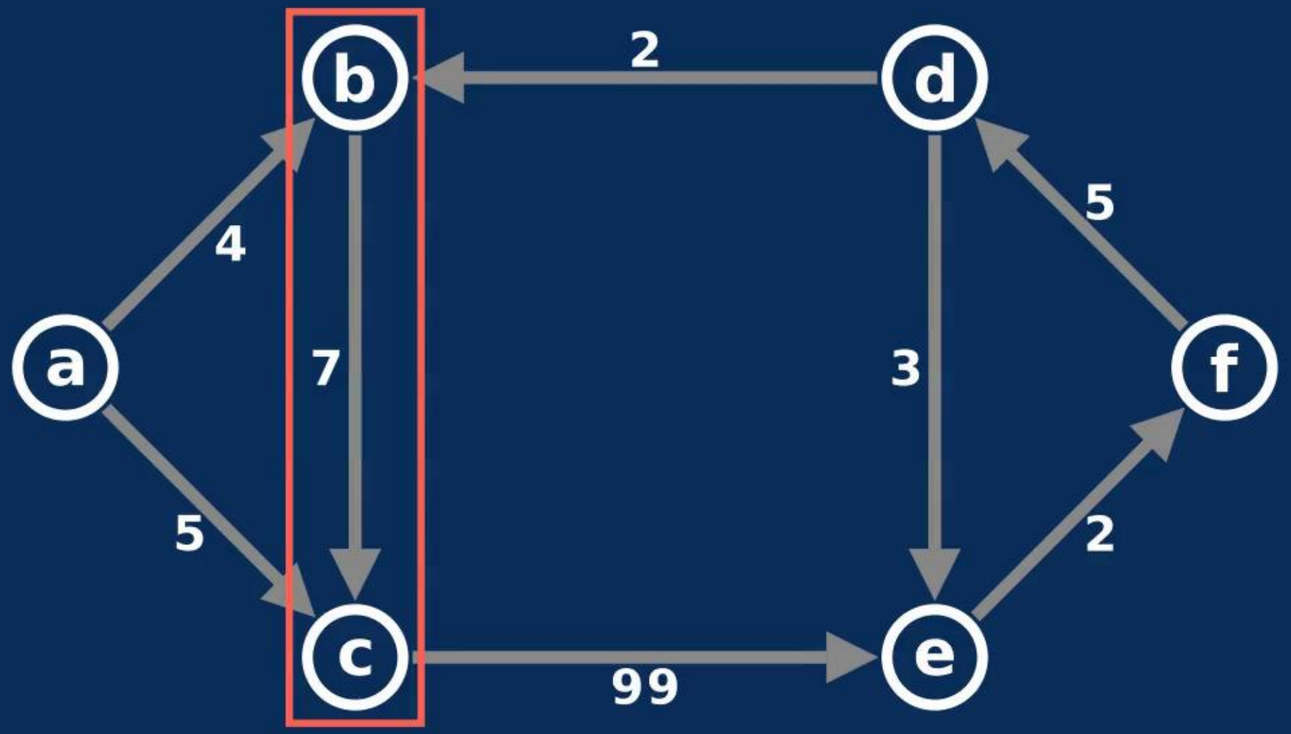
Introduction

	a	b	c	d	e	f
a	0	4	5	0	0	0
b	0	0	7	0	0	0
c	0	0	0	0	99	0
d	0	2	0	0	3	0
e	0	0	0	0	0	2
f	0	0	0	5	0	0



Row is source, Column is destination

	a	b	c	d	e	f
a	0	4	5	0	0	0
b	0	0	7	0	0	0
c	0	0	0	0	99	0
d	0	2	0	0	3	0
e	0	0	0	0	0	2
f	0	0	0	5	0	0

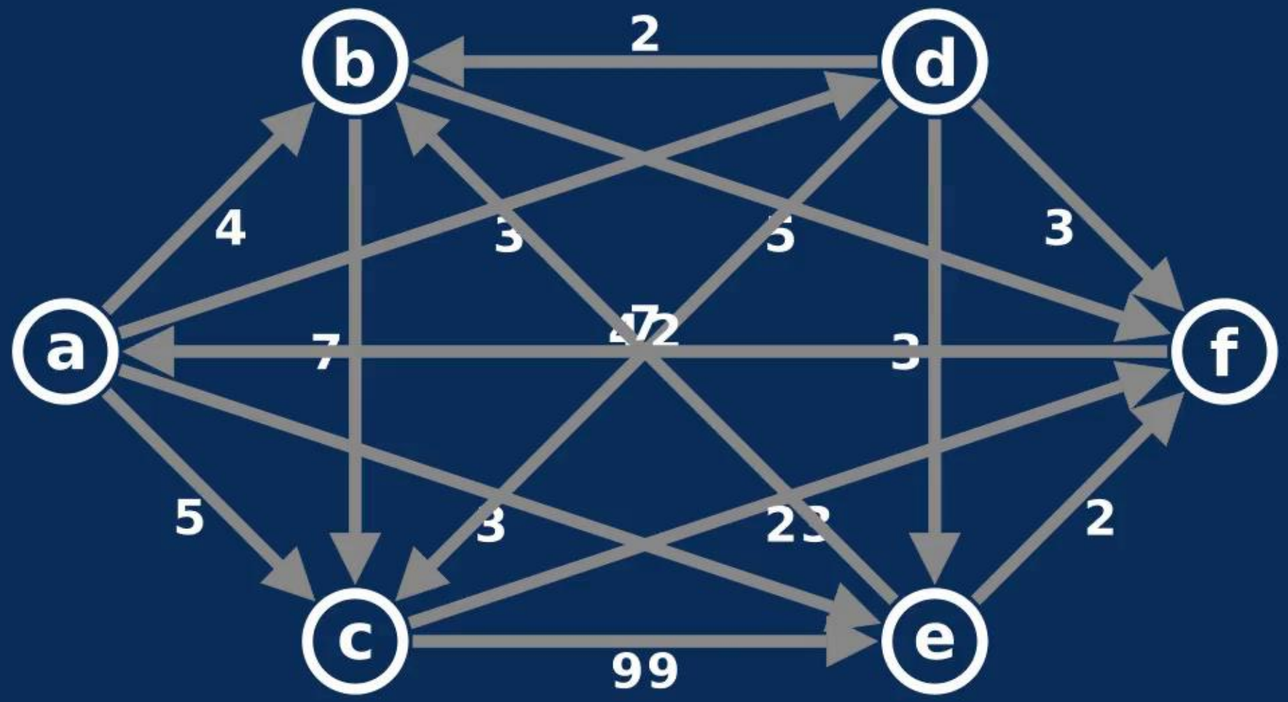


Row is source, Column is destination

What kind of graph would you use this for?

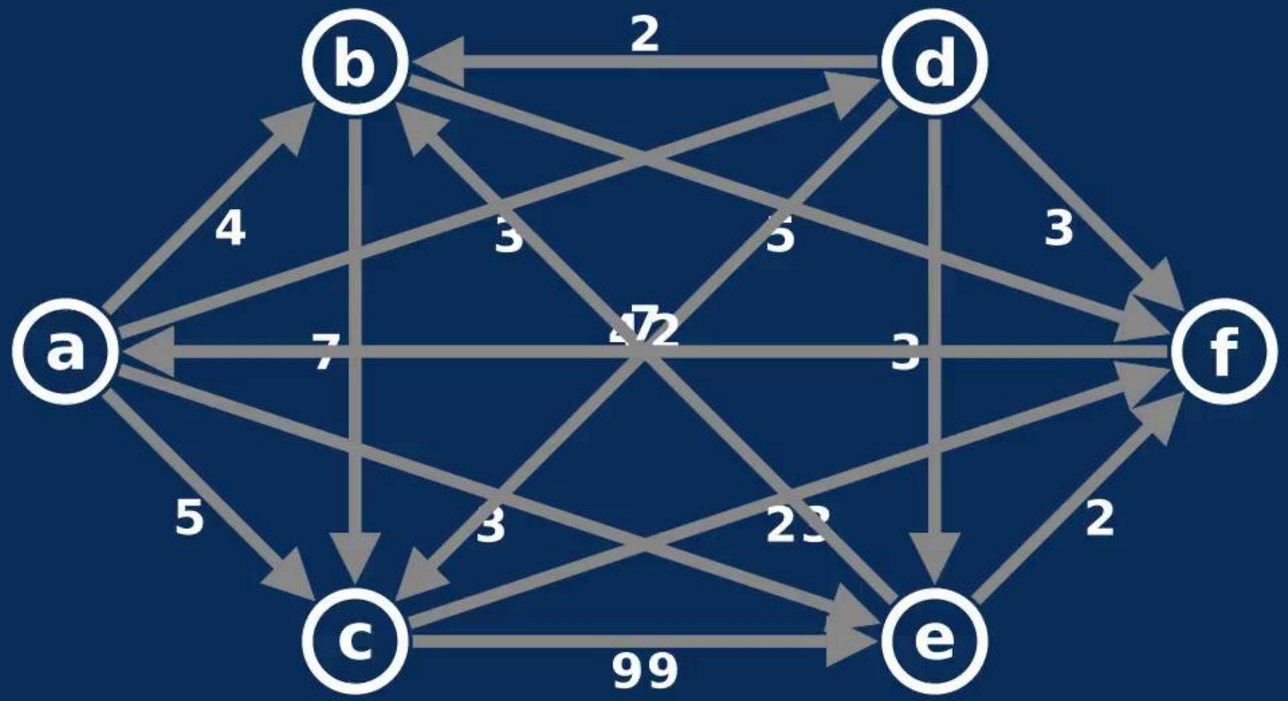
Dense Graph

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	0	0	0	0	0	0
c	0	0	0	0	0	0
d	0	0	0	0	0	0
e	0	0	0	0	0	0
f	0	0	0	0	0	0



Dense Graph

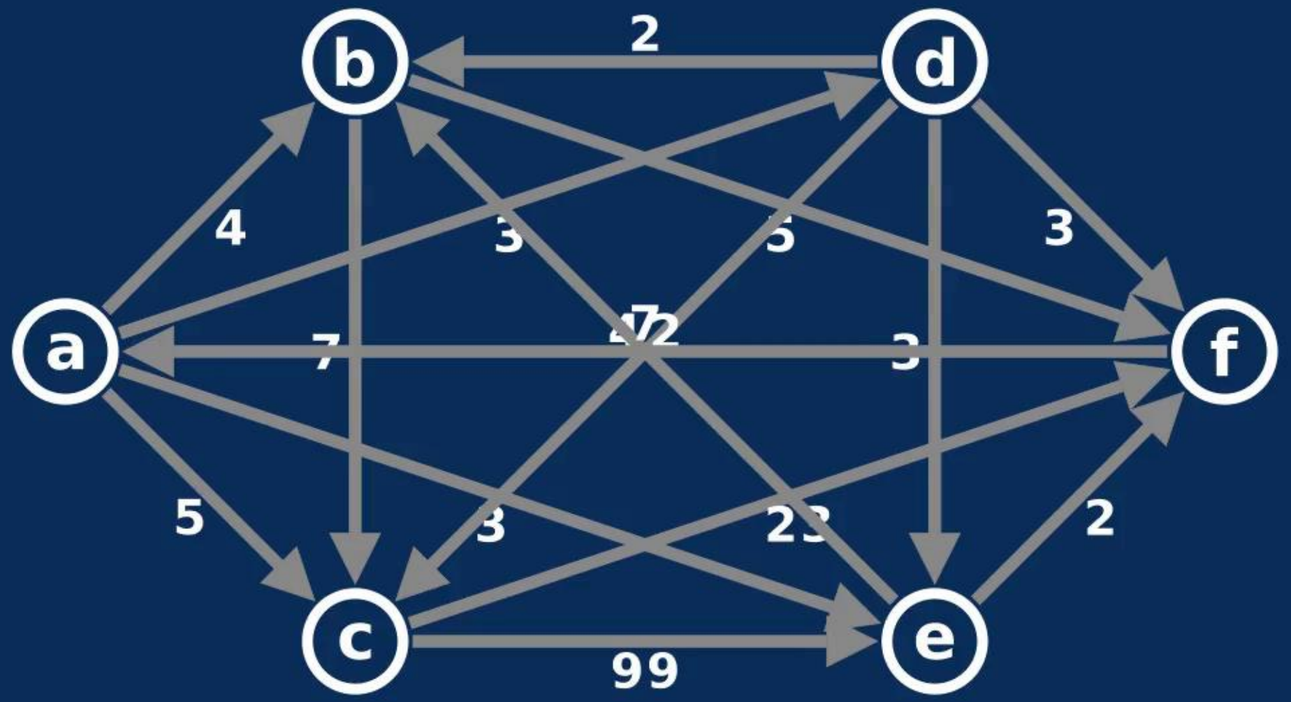
	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0



Dense graphs work well with this, since ‘most’ pairs are connected.

Dense Graph

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

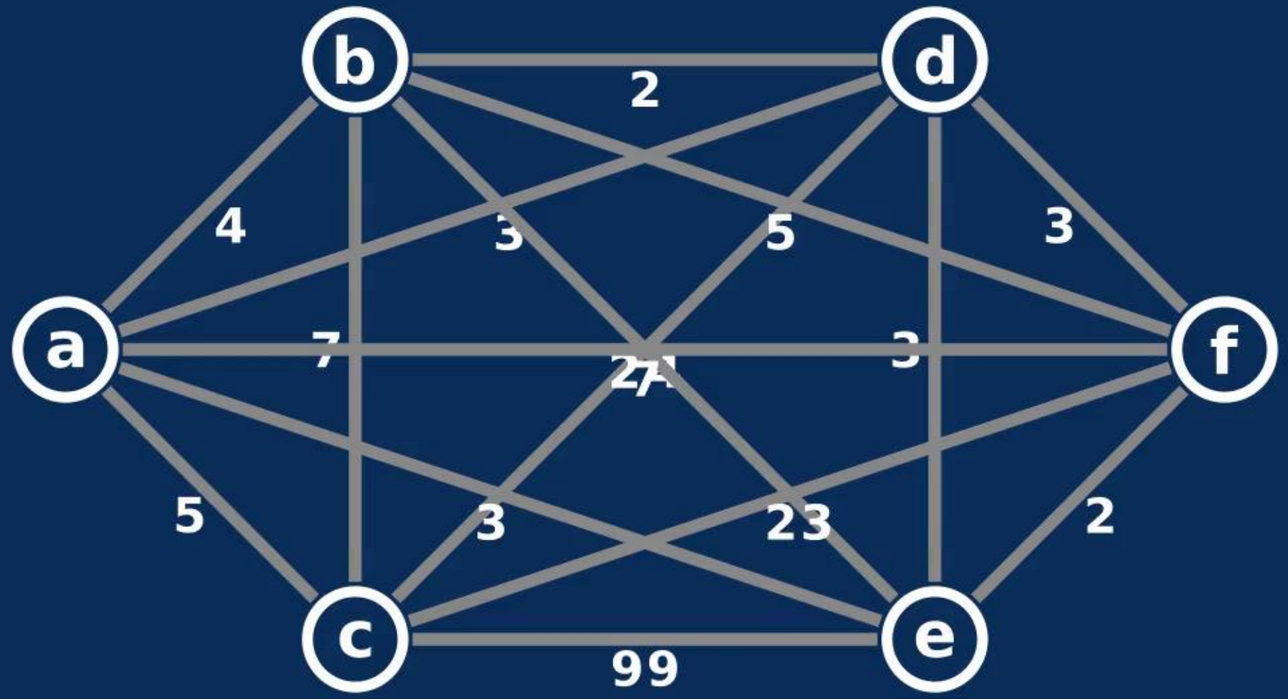


Dense graphs work well with this, since ‘most’ pairs are connected.

Let’s make this graph undirected! What will happen?

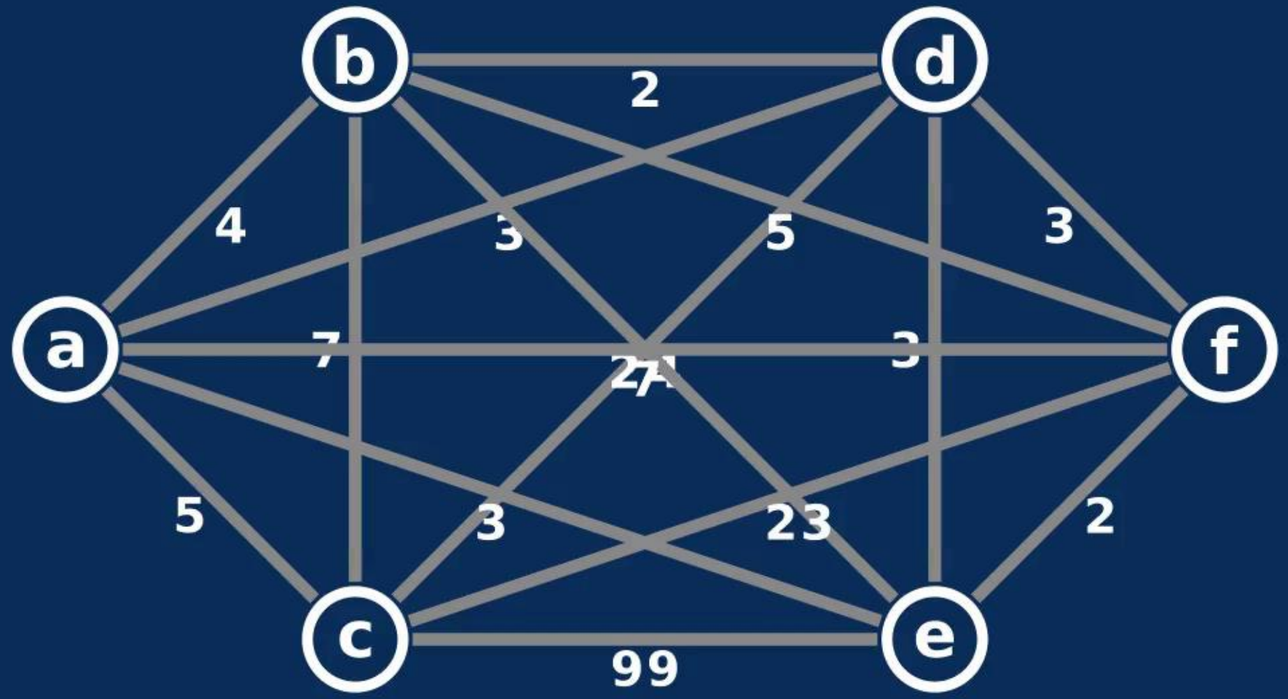
Undirected Graph Adjacency Matrix

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	0	0	0	0	0	0
c	0	0	0	0	0	0
d	0	0	0	0	0	0
e	0	0	0	0	0	0
f	0	0	0	0	0	0



Undirected Graph Adjacency Matrix

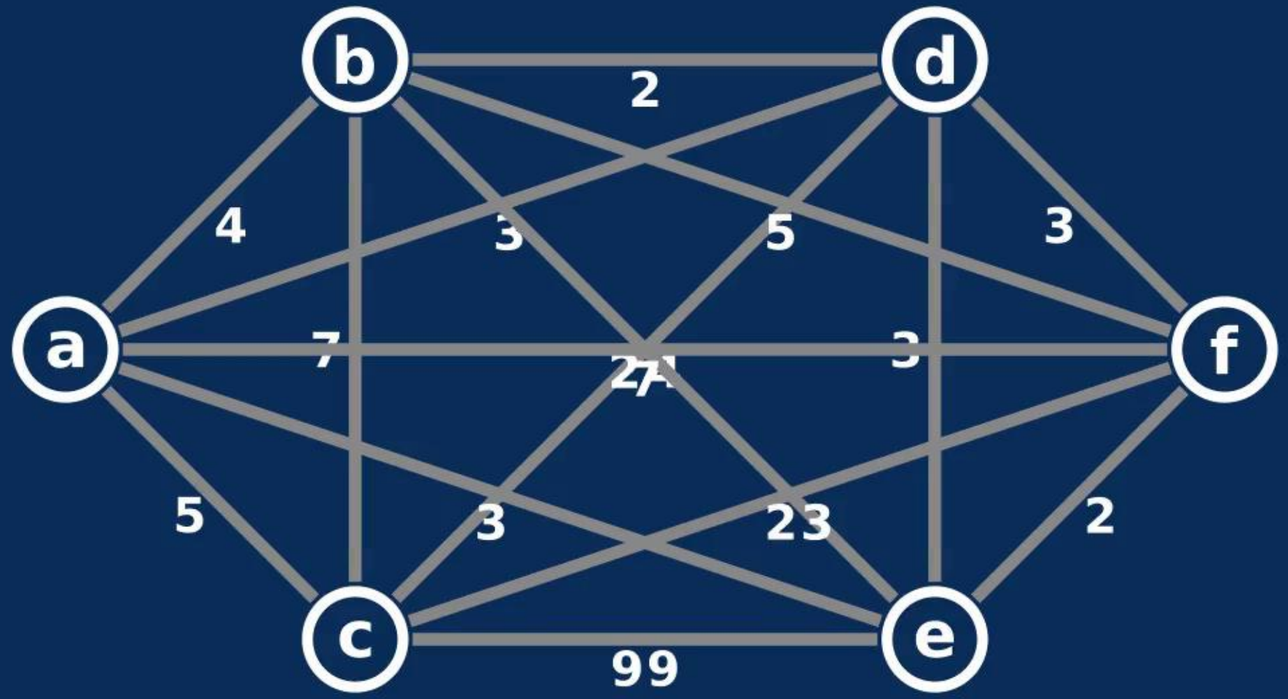
	a	b	c	d	e	f
a	0	4	5	3	3	7
b	4	0	7	2	2	5
c	5	7	0	4	99	23
d	3	2	4	0	3	3
e	3	2	99	3	0	2
f	7	5	23	3	2	0



What do you notice? Can we optimize this?

Undirected Graph Adjacency Matrix

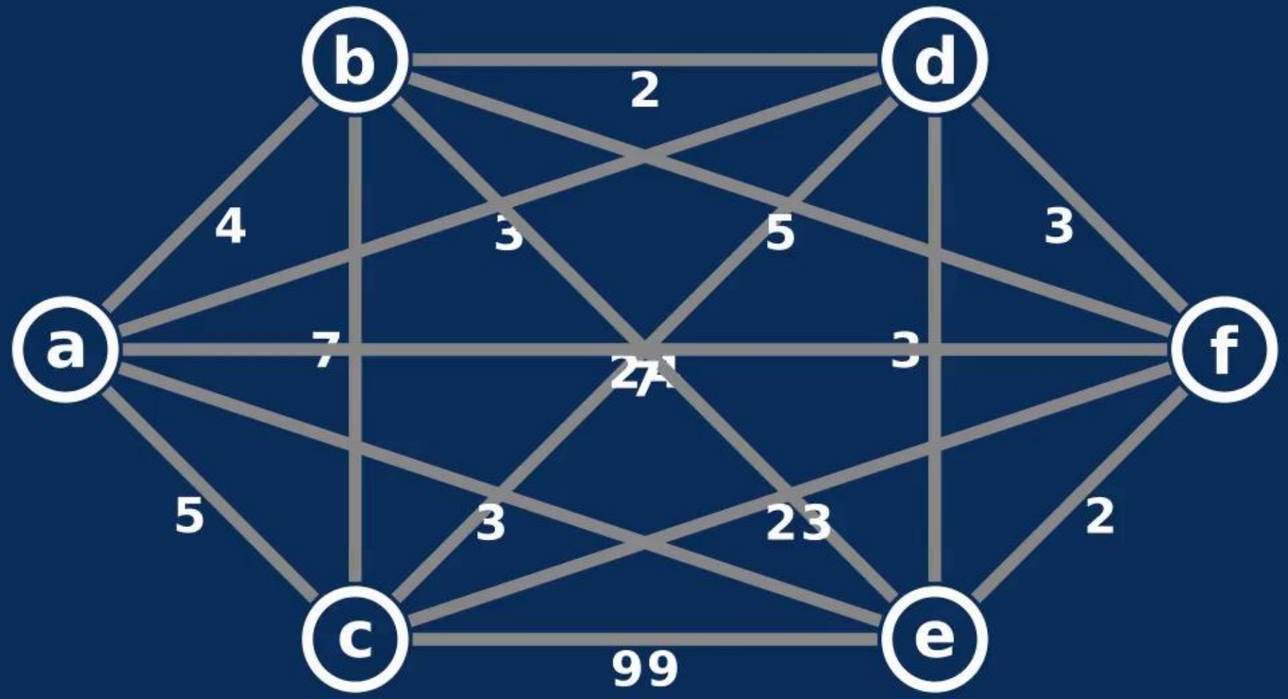
	a	b	c	d	e	f
a	0	4	5	3	3	7
b	4	0	7	2	2	5
c	5	7	0	4	99	23
d	3	2	4	0	3	3
e	3	2	99	3	0	2
f	7	5	23	3	2	0



What do you notice? Can we optimize this?

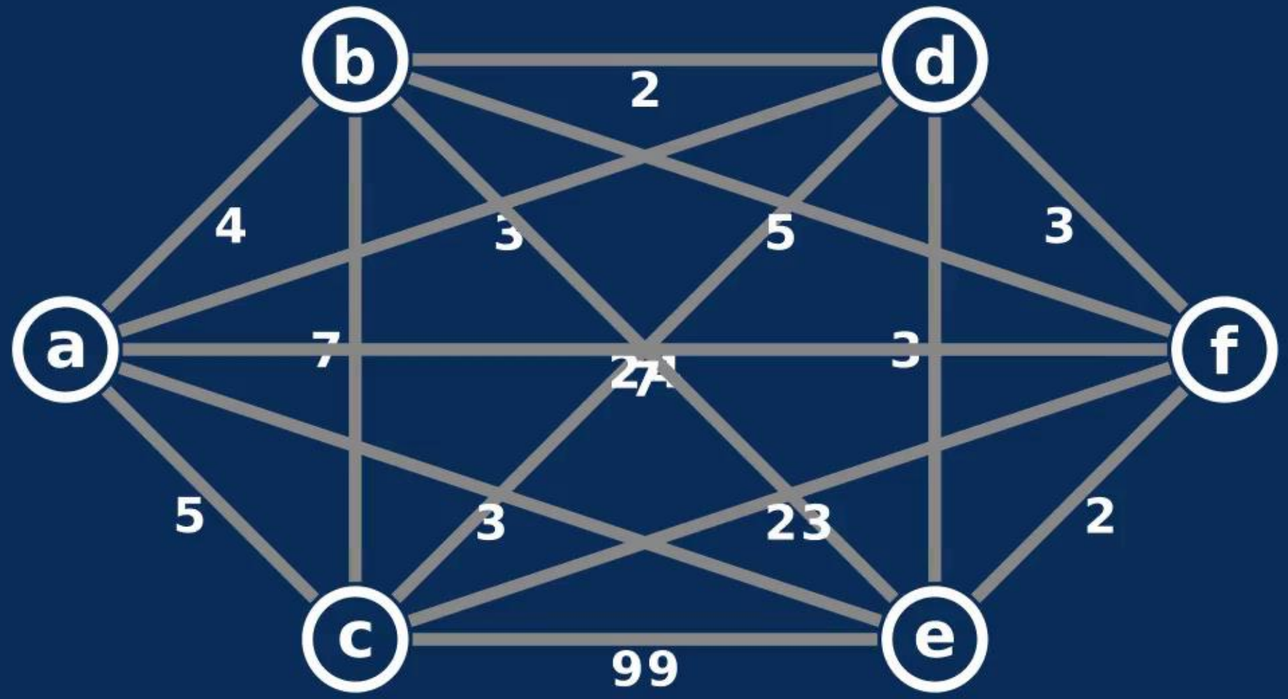
Triangular Matrix Adjacency Matrix

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	0	0	0	0	0	0
c	0	0	0	0	0	0
d	0	0	0	0	0	0
e	0	0	0	0	0	0
f	0	0	0	0	0	0



Triangular Matrix Adjacency Matrix

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	0	0	0	0	0	0
c	0	0	0	0	0	0
d	0	0	0	0	0	0
e	0	0	0	0	0	0
f	0	0	0	0	0	0



You want a lower triangular matrix...

Triangular Matrix Adjacency Matrix

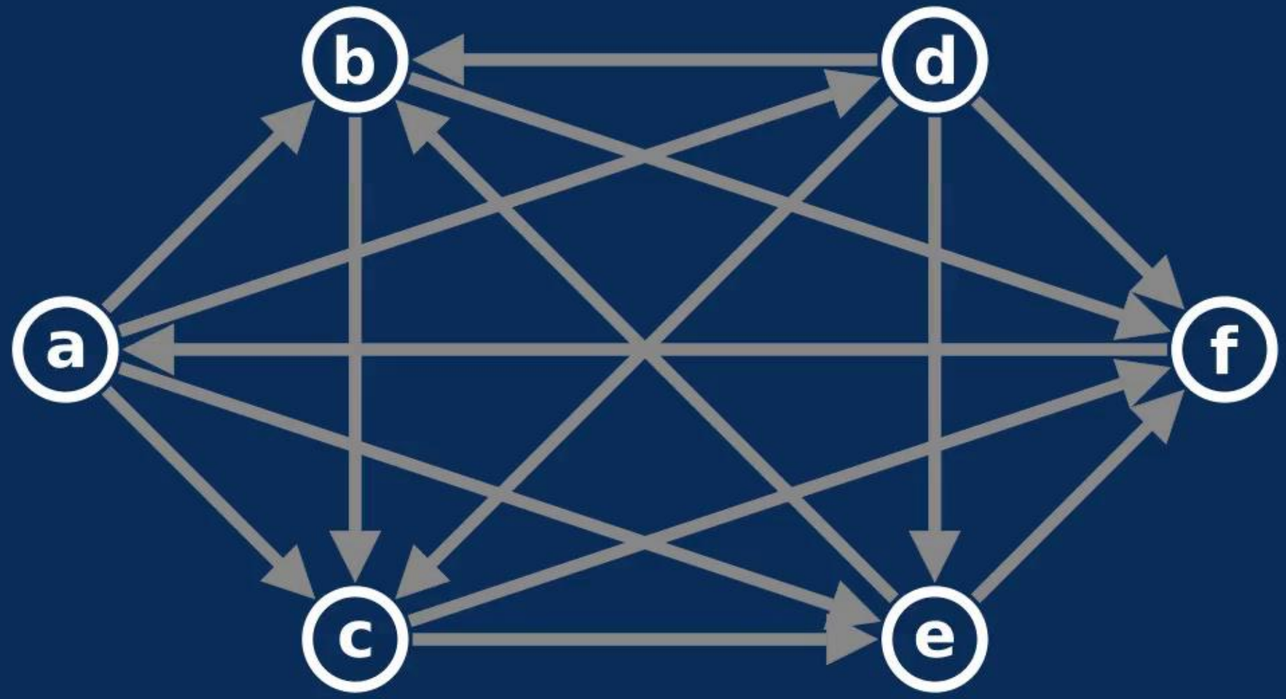
	a	b	c	d	e	f
a	0	0	0	0	0	0
b	4	0	0	0	0	0
c	5	7	0	0	0	0
d	3	2	4	0	0	0
e	3	2	99	3	0	0
f	7	5	23	3	2	0

```
1 vector<vector<int>> adj_mat;
2 for(int i=0; i<n; ++i)
3     adj_mat.push_back(vector<int>(i+1))
```

You want a lower triangular matrix...

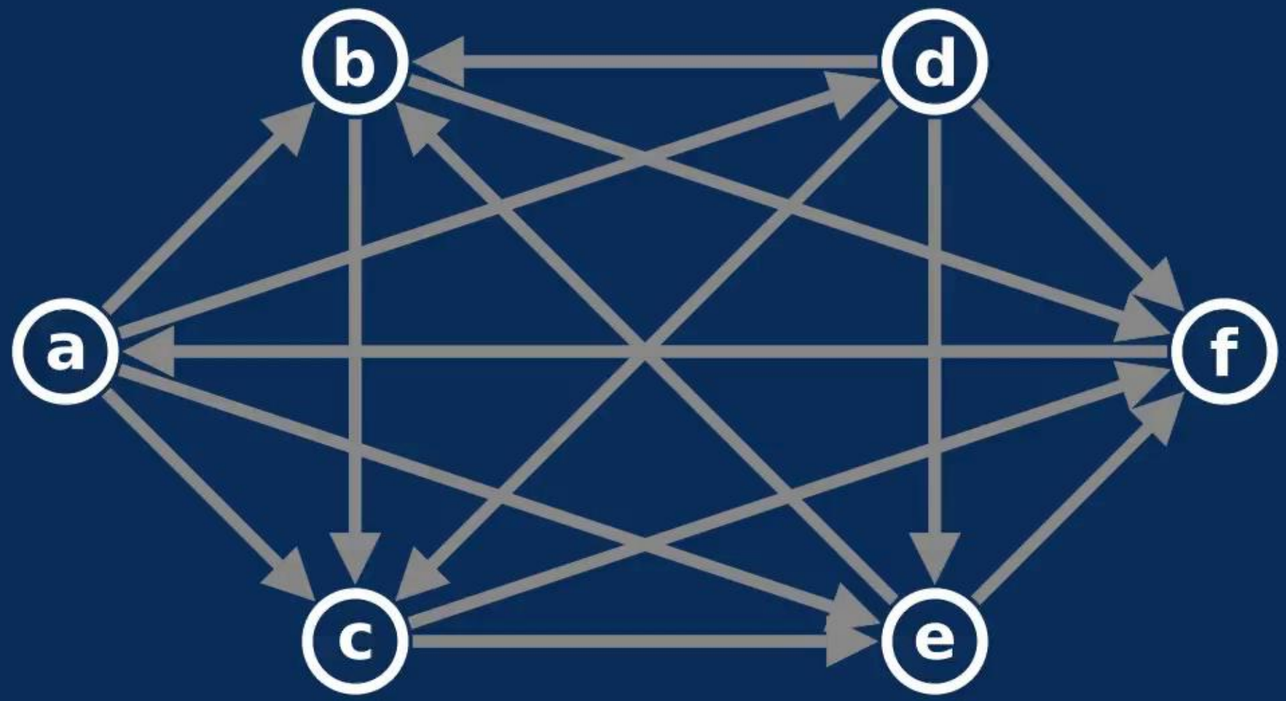
Unweighted Graph Adjacency Matrix

	a	b	c	d	e	f
a	0	0	0	0	0	0
b	0	0	0	0	0	0
c	0	0	0	0	0	0
d	0	0	0	0	0	0
e	0	0	0	0	0	0
f	0	0	0	0	0	0



Unweighted Graph Adjacency Matrix

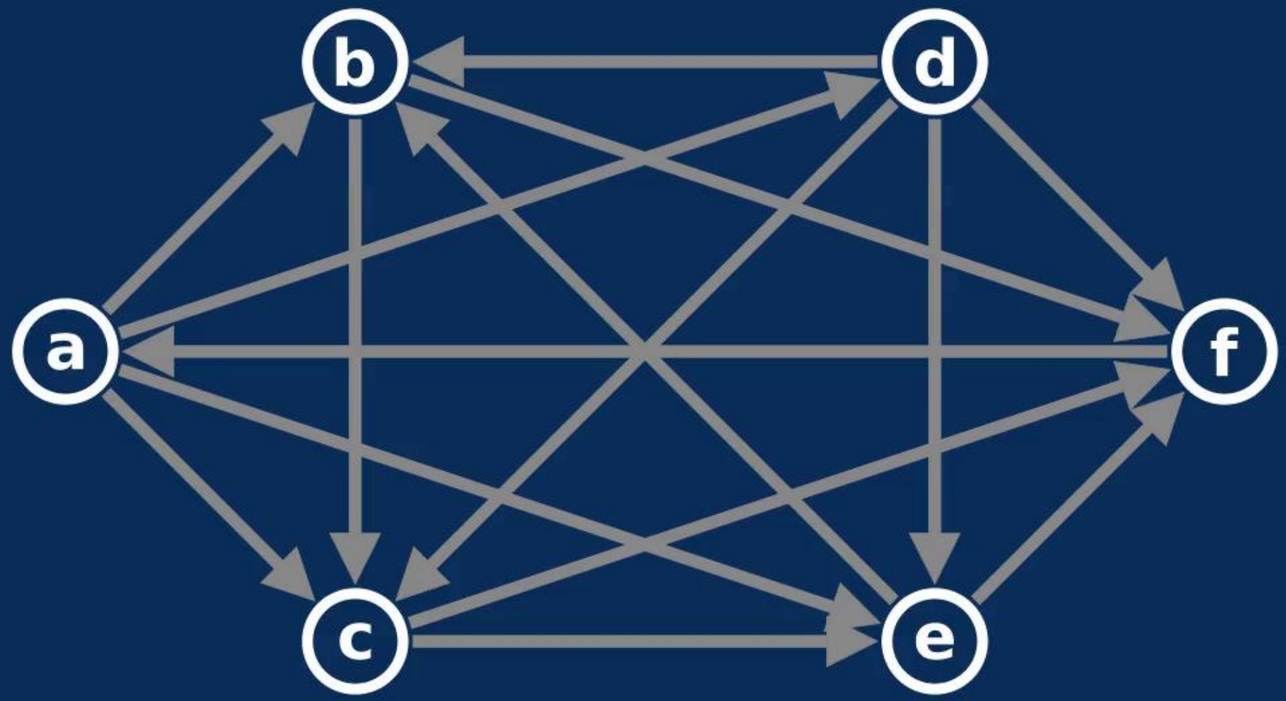
	a	b	c	d	e	f
a	0	1	1	1	1	0
b	0	0	1	0	0	1
c	0	0	0	0	1	1
d	0	1	1	0	1	1
e	0	1	0	0	0	1
f	1	0	0	0	0	0



If the graph is unweighted we have another option.

Unweighted Graph Adjacency Matrix

	a	b	c	d	e	f
a	0	1	1	1	1	0
b	0	0	1	0	0	1
c	0	0	0	0	1	1
d	0	1	1	0	1	1
e	0	1	0	0	0	1
f	1	0	0	0	0	0



If the graph is unweighted we have another option.

We can use bit vectors!

Graph ADT

Modification

- `insertVertex(K key)`
- `insertEdge(Vertex v1, Vertex v2, K Key)`
- `removeVertex(K key)`
- `removeEdge(Vertex v1, Vertex v2)`

Need storage for vertices and keys. Maybe weights.

Graph ADT

Query

- `getEdges(Vertex v)`
- `areAdjecent(Vertex v1,Vertex v2)`
- `origin(Edge e)`
- `destination(Edge e)`

Timing

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`insertVertex(K key)`

`insertEdge(Vertex v1, Vertex v2, K Key)`

`removeVertex(K key)`

`removeEdge(Vertex v1, Vertex v2)`

Timing

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`insertVertex(K key)`

- $\mathcal{O}(1)$ (if you planned ahead)

`insertEdge(Vertex v1, Vertex v2, K Key)`

`removeVertex(K key)`

`removeEdge(Vertex v1, Vertex v2)`

Timing

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`insertVertex(K key)`

- $\mathcal{O}(1)$ (if you planned ahead)

`insertEdge(Vertex v1, Vertex v2, K Key)`

- $\mathcal{O}(1)$

`removeVertex(K key)`

`removeEdge(Vertex v1, Vertex v2)`

Timing

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`insertVertex(K key)`

- $\mathcal{O}(1)$ (if you planned ahead)

`insertEdge(Vertex v1, Vertex v2, K Key)`

- $\mathcal{O}(1)$

`removeVertex(K key)`

- !!!

`removeEdge(Vertex v1, Vertex v2)`

Timing

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`insertVertex(K key)`

- $\mathcal{O}(1)$ (if you planned ahead)

`insertEdge(Vertex v1, Vertex v2, K Key)`

- $\mathcal{O}(1)$

`removeVertex(K key)`

- !!!
- $\mathcal{O}(1)$ You just annotate vertex list

`removeEdge(Vertex v1, Vertex v2)`

Timing

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`insertVertex(K key)`

- $\mathcal{O}(1)$ (if you planned ahead)

`insertEdge(Vertex v1, Vertex v2, K Key)`

- $\mathcal{O}(1)$

`removeVertex(K key)`

- !!!
- $\mathcal{O}(1)$ You just annotate vertex list

`removeEdge(Vertex v1, Vertex v2)`

- $\mathcal{O}(1)$

Timing, 2

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`getEdges(Vertex v)`

`areAdjacent(Vertex v1, Vertex v2)`

`origin(Edge e)`

Timing, 2

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`getEdges(Vertex v)`

- $\mathcal{O}(n)$ - Need to traverse array

`areAdjacent(Vertex v1, Vertex v2)`

`origin(Edge e)`

Timing, 2

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`getEdges(Vertex v)`

- $\mathcal{O}(n)$ - Need to traverse array

`areAdjacent(Vertex v1, Vertex v2)`

- $\mathcal{O}(1)$

`origin(Edge e)`

Timing, 2

	a	b	c	d	e	f
a	0	4	5	3	3	0
b	0	0	7	0	0	5
c	0	0	0	0	99	23
d	0	2	4	0	3	3
e	0	2	0	0	0	2
f	7	0	0	0	0	0

How long do things take?

`getEdges(Vertex v)`

- $\mathcal{O}(n)$ - Need to traverse array

`areAdjacent(Vertex v1, Vertex v2)`

- $\mathcal{O}(1)$

`origin(Edge e)`

- $\mathcal{O}(1)$