Graph representations
COMS20017 (Algorithms and Data)

John Lapinskas, University of Bristol

Adjacency matrices

I haven't talked at all about running times of graph algorithms...

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

Adjacency matrices

I haven't talked at all about running times of graph algorithms...

because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$

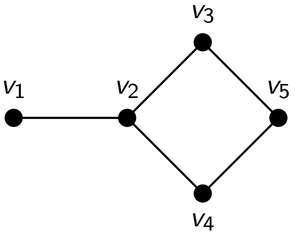
Adjacency matrices

I haven't talked at all about running times of graph algorithms...

because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



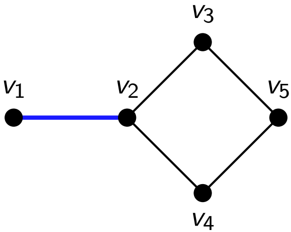
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \dots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



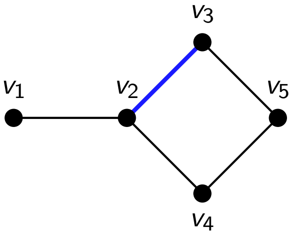
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \dots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



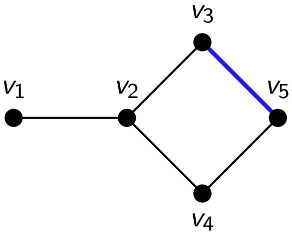
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & \mathbf{1} & 1 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



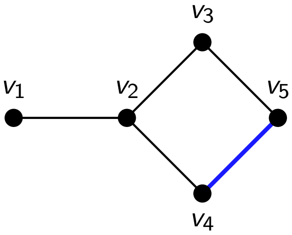
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & \mathbf{1} \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & \mathbf{1} & 1 & 0 \end{pmatrix}$$

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 1 & \mathbf{1} & 0 \end{pmatrix}$$

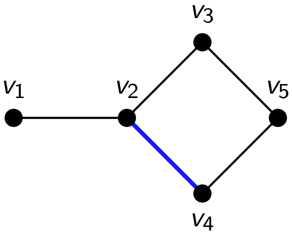
Adjacency matrices

I haven't talked at all about running times of graph algorithms...

because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



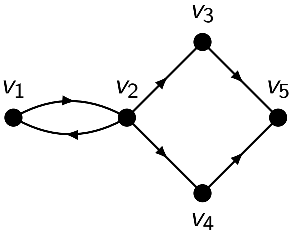
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & \mathbf{1} & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & \mathbf{1} & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



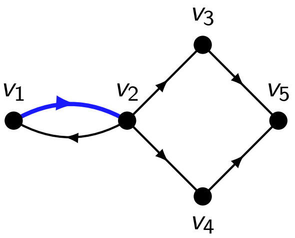
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



$$\begin{pmatrix} 0 & \mathbf{1} & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

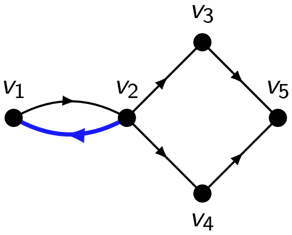
Adjacency matrices

I haven't talked at all about running times of graph algorithms...

because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ \mathbf{1} & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

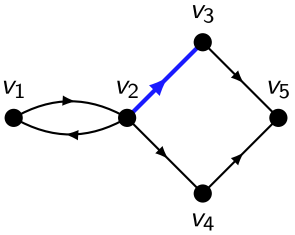
Adjacency matrices

I haven't talked at all about running times of graph algorithms...

because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & \mathbf{1} & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

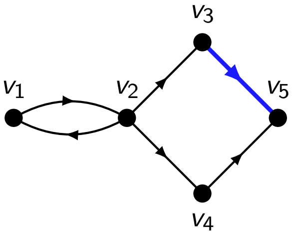
Adjacency matrices

I haven't talked at all about running times of graph algorithms...

because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



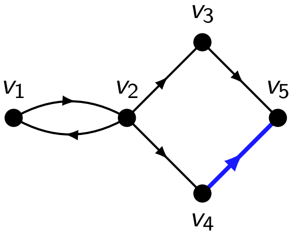
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



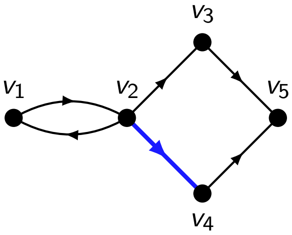
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \dots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



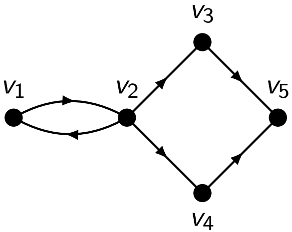
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Storing the matrix as a 2D array takes $\mathbf{\Theta(|V|^2)}$ **space**.

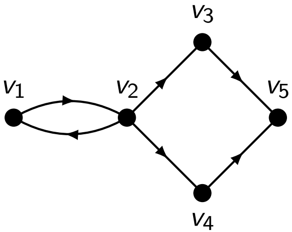
Adjacency matrices

I haven't talked at all about running times of graph algorithms...

because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

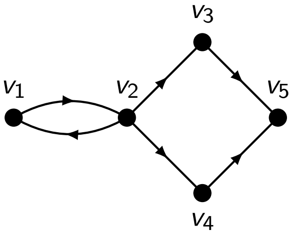
Storing the matrix as a 2D array takes $\mathbf{\Theta(|V|^2)}$ **space**.
An **adjacency query** ("Is $(u, v) \in E$?") takes $\mathbf{\Theta(1)}$ **time**.

Adjacency matrices

I haven't talked at all about running times of graph algorithms...
because a lot depends on how the input graphs are stored.

One popular way to store a graph $G = (V, E)$ is an **adjacency matrix**:

$$V =: \{v_1, v_2, \ldots, v_n\}, \qquad A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j, \\ 0 & \text{otherwise.} \end{cases}$$



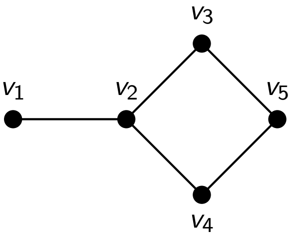
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

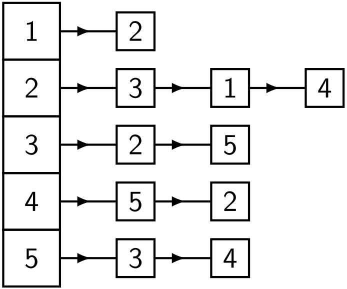
Storing the matrix as a 2D array takes $\boldsymbol{\Theta(|V|^2)}$ **space**.
An **adjacency query** ("Is $(u, v) \in E$?") takes $\boldsymbol{\Theta(1)}$ **time**.
A **neighbourhood query** ("What is $N^+(u)$?") takes $\boldsymbol{\Theta(|V|)}$ **time**.

Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

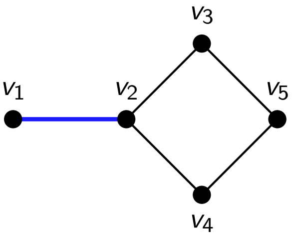


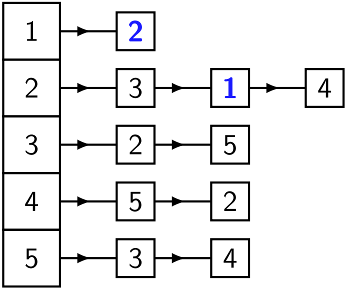


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \dots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

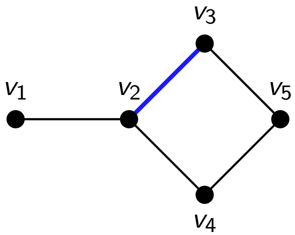


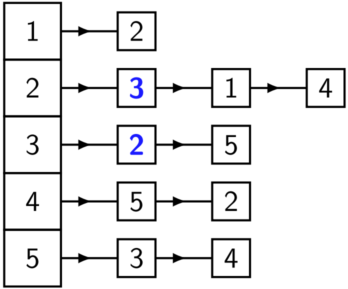


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

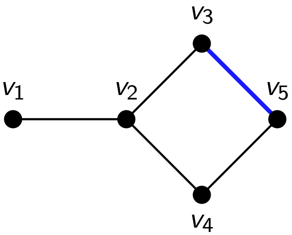


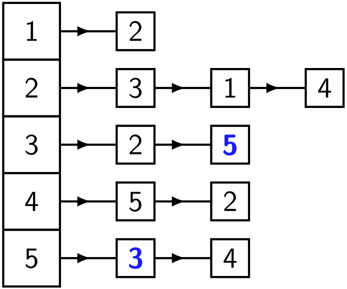


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

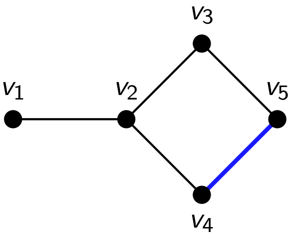


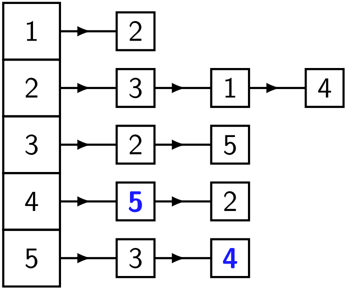


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

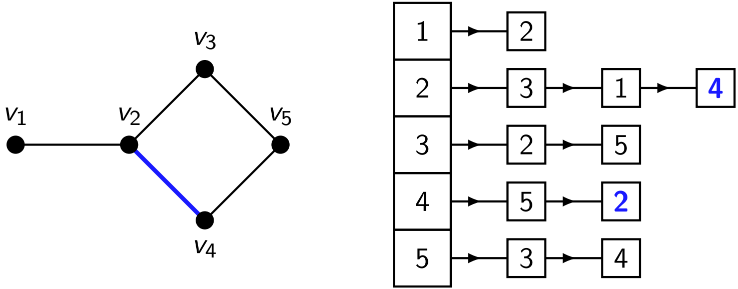




Adjacency lists

Another way is **adjacency list format**.

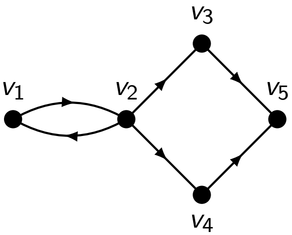
Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

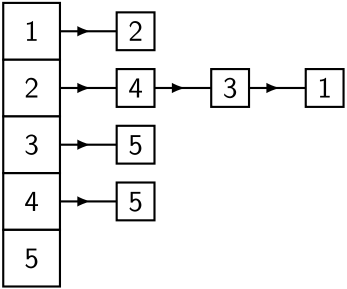


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

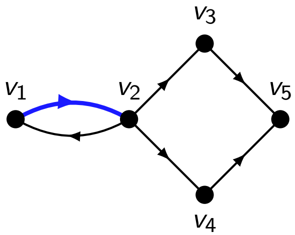


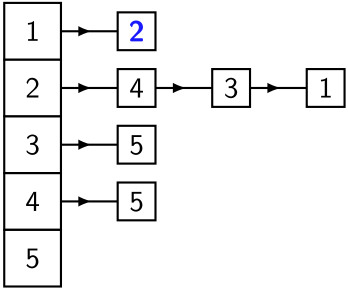


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

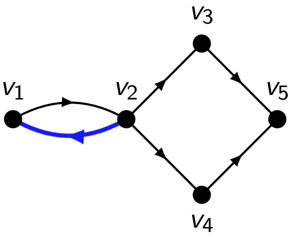


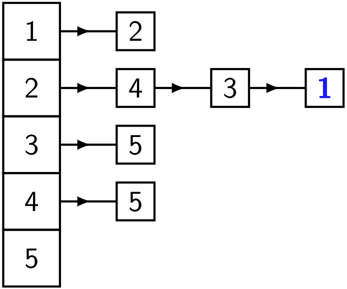


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

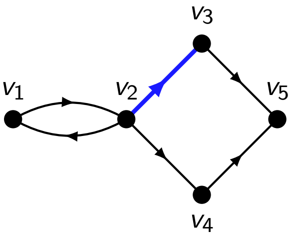


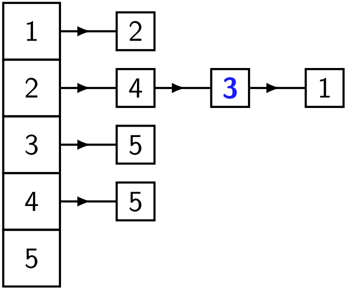


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \dots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

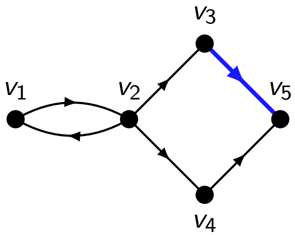


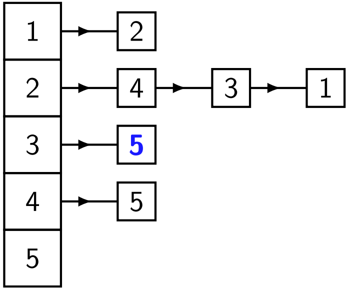


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

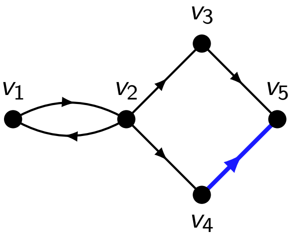


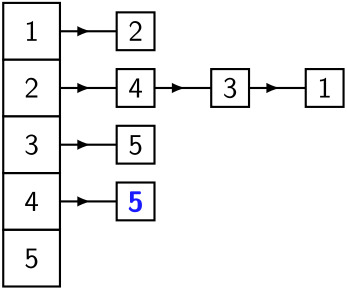


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

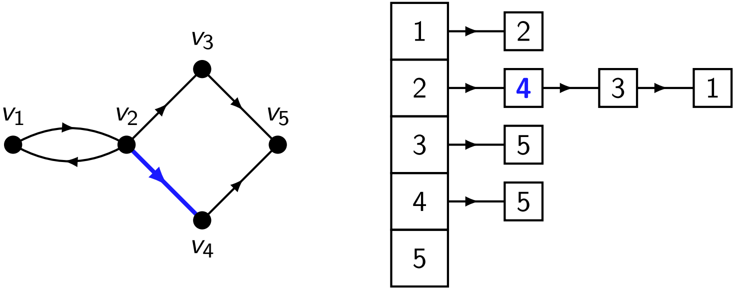




Adjacency lists

Another way is **adjacency list format**.

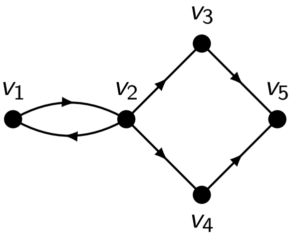
Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):

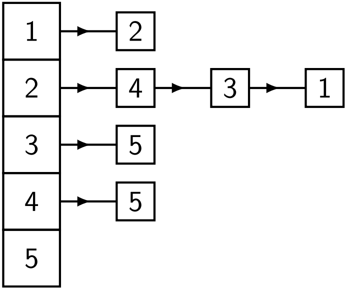


Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \dots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):



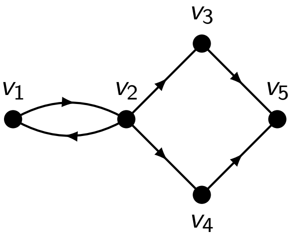


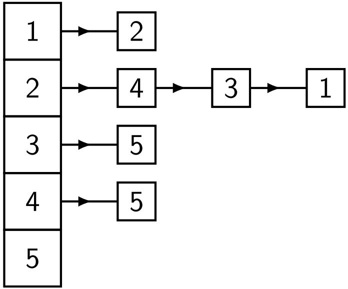
Storing the array of lists takes $\mathbf{\Theta(|V| + |E|)}$ **space**.

Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):



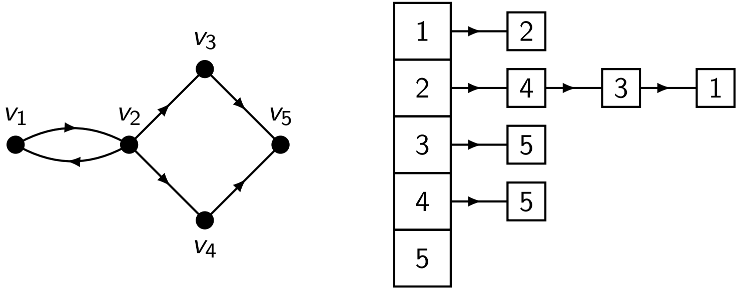


Storing the array of lists takes $\boldsymbol{\Theta(|V| + |E|)}$ **space**.
An **adjacency query** ("Is $(u, v) \in E$?") takes $\boldsymbol{\Theta(d^+(u))}$ **time**.

Adjacency lists

Another way is **adjacency list format**.

Here we store an array of $|V|$ linked lists $L_1, \ldots, L_n$, where L_i contains the list of vertices v_j with $(v_i, v_j) \in E$ (in some order):



Storing the array of lists takes $\boldsymbol{\Theta(|V| + |E|)}$ **space**.
An **adjacency query** (“Is $(u, v) \in E$?”) takes $\boldsymbol{\Theta(d^+(u))}$ **time**.
A **neighbourhood query** (“What is $N^+(u)$?”) takes $\boldsymbol{\Theta(d^+(u))}$ **time**.

Matrices or lists?

Advantages of matrices	Advantages of lists
Fast adjacency queries for all graphs	Fast degree queries for sparse graphs

Matrices or lists?

Advantages of matrices	Advantages of lists
Fast adjacency queries for all graphs	Fast degree queries for sparse graphs
Can sometimes use linear algebra for fast algorithms (see problem sheet)	Low space requirement for sparse graphs

Matrices or lists?

Advantages of matrices	**Advantages of lists**
Fast adjacency queries for all graphs	Fast degree queries for sparse graphs
Can sometimes use linear algebra for fast algorithms (see problem sheet)	Low space requirement for sparse graphs
	Can drop adjacency queries to $O(1)$ **expected** time with hash tables (c.f. Advanced Algorithms next year!)

Matrices or lists?

Advantages of matrices	Advantages of lists
Fast adjacency queries for all graphs	Fast degree queries for sparse graphs
Can sometimes use linear algebra for fast algorithms (see problem sheet)	Low space requirement for sparse graphs
	Can drop adjacency queries to $O(1)$ **expected** time with hash tables (c.f. Advanced Algorithms next year!)

In practice, we normally use adjacency lists with hash tables over adjacency matrices or vanilla adjacency lists.

Matrices or lists?

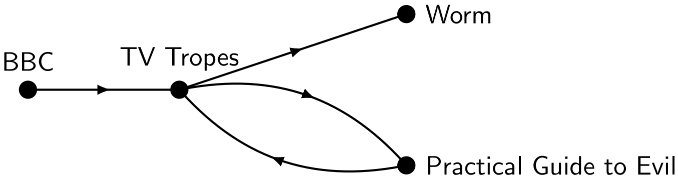
Advantages of matrices	Advantages of lists
Fast adjacency queries for all graphs	Fast degree queries for sparse graphs
Can sometimes use linear algebra for fast algorithms (see problem sheet)	Low space requirement for sparse graphs
	Can drop adjacency queries to $O(1)$ **expected** time with hash tables (c.f. Advanced Algorithms next year!)

In practice, we normally use adjacency lists with hash tables over adjacency matrices or vanilla adjacency lists... **unless** we aren't actually storing the graph at all!

Implicit graphs

A key ingredient of Google's search algorithm is **PageRank**: a rough indicator of the "importance" of a given site on the Internet, computed by looking at incoming and outgoing links.

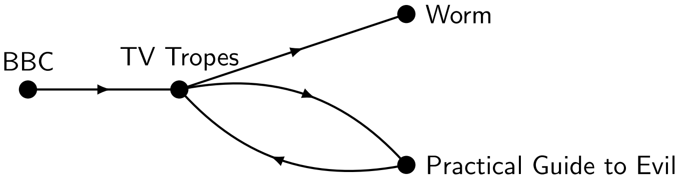
Calculating PageRank is a **graph problem**:



Implicit graphs

A key ingredient of Google's search algorithm is **PageRank**: a rough indicator of the "importance" of a given site on the Internet, computed by looking at incoming and outgoing links.

Calculating PageRank is a **graph problem**:

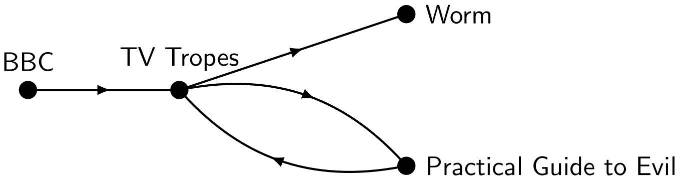


Google **really** doesn't want to have to store the graph.

Implicit graphs

A key ingredient of Google's search algorithm is **PageRank**: a rough indicator of the "importance" of a given site on the Internet, computed by looking at incoming and outgoing links.

Calculating PageRank is a **graph problem**:



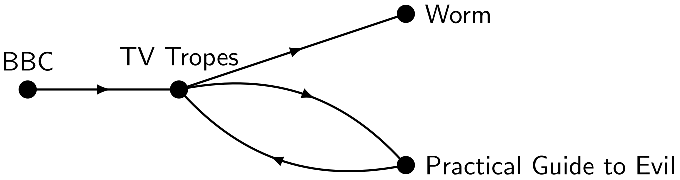
Google **really** doesn't want to have to store the graph.

But they can easily list neighbours of a site v in $O(d(v))$ time...

Implicit graphs

A key ingredient of Google's search algorithm is **PageRank**: a rough indicator of the "importance" of a given site on the Internet, computed by looking at incoming and outgoing links.

Calculating PageRank is a **graph problem**:



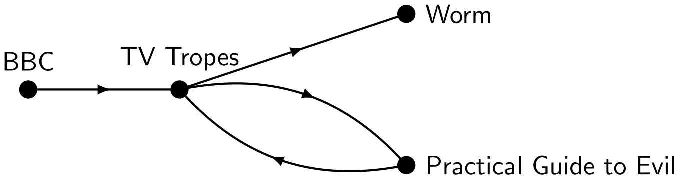
Google **really** doesn't want to have to store the graph.

But they can easily list neighbours of a site v in $O(d(v))$ time...
so they can have *the same interface* that adjacency lists would provide, but without actually storing anything!

Implicit graphs

A key ingredient of Google's search algorithm is **PageRank**: a rough indicator of the "importance" of a given site on the Internet, computed by looking at incoming and outgoing links.

Calculating PageRank is a **graph problem**:



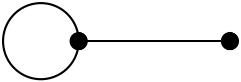
Google <u>**really**</u> doesn't want to have to store the graph.

But they can easily list neighbours of a site v in $O(d(v))$ time...
so they can have *the same interface* that adjacency lists would provide, but without actually storing anything!

Situations like this, where the graph is only stored implicitly, are why we really care about the adjacency list and matrix models.

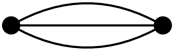
Loops and multiple edges

Sometimes it's useful to consider graphs with:



Loops

connecting vertices to themselves;



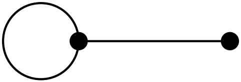
Multiple edges

between one pair of vertices.

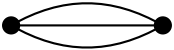
These are called **multigraphs**.

Loops and multiple edges

Sometimes it's useful to consider graphs with:



Loops
connecting vertices to themselves;



Multiple edges
between one pair of vertices.

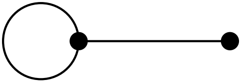
These are called **multigraphs**.

Results and algorithms for graphs usually carry over to multigraphs unchanged, but they often make things harder to visualise and write.

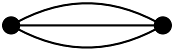
And loops and multiple edges are rarely necessary.

Loops and multiple edges

Sometimes it's useful to consider graphs with:



Loops connecting vertices to themselves;



Multiple edges between one pair of vertices.

These are called **multigraphs**.

Results and algorithms for graphs usually carry over to multigraphs unchanged, but they often make things harder to visualise and write.

And loops and multiple edges are rarely necessary.

So in this course we will only consider standard (a.k.a. **simple**) graphs, without loops or multiple edges.