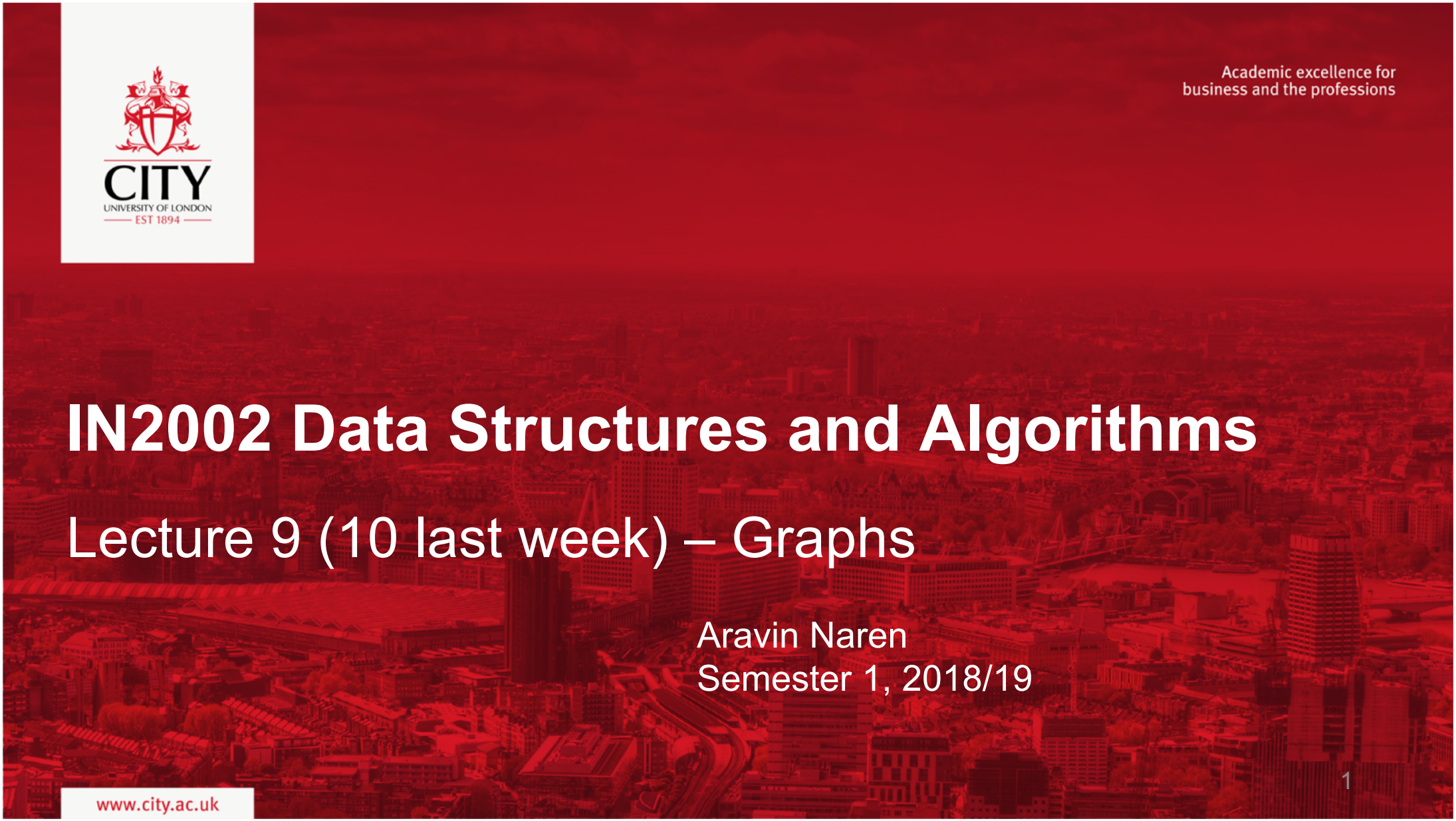

CITY
UNIVERSITY OF LONDON
EST 1894
Academic excellence for
business and the professions
IN2002 Data Structures and Algorithms
Lecture 9 (10 last week) – Graphs
Aravin Naren
Semester 1, 2018/19
www.city.ac.uk
1

Learning Objectives

- Understand and be able to use the data structures graphs

- Be able to understand, apply and develop algorithms to handle graphs. Including:
  - Traversals
  - Shortest paths

Trees and Graphs

Trees and graphs

- Trees represent a set of elements and their relations.
- Each element is stored in a node.
- Relations go from parent to child nodes.
- A node can have at most one parent.

- Graphs extend the idea by removing the parent-child restriction.

Nodes and links between nodes

- In heaps, the links represent the fact that child nodes have lower values than their parents.

  In binary search trees, left links represent the fact that all values in the left subtree are less than its parent, and all values in the right subtree are greater than it.

- The value of a node and the knowledge of what its branches represent allows guiding algorithms.

Non-hierarchical relations

- Trees allow a node only to have one parent.
- Nodes may be related in the same way to different parent nodes (e.g. family tree, web pages).
- A situation associated with this are loops, which usually breaks tree algorithms.
- Graphs provide a more general structure.

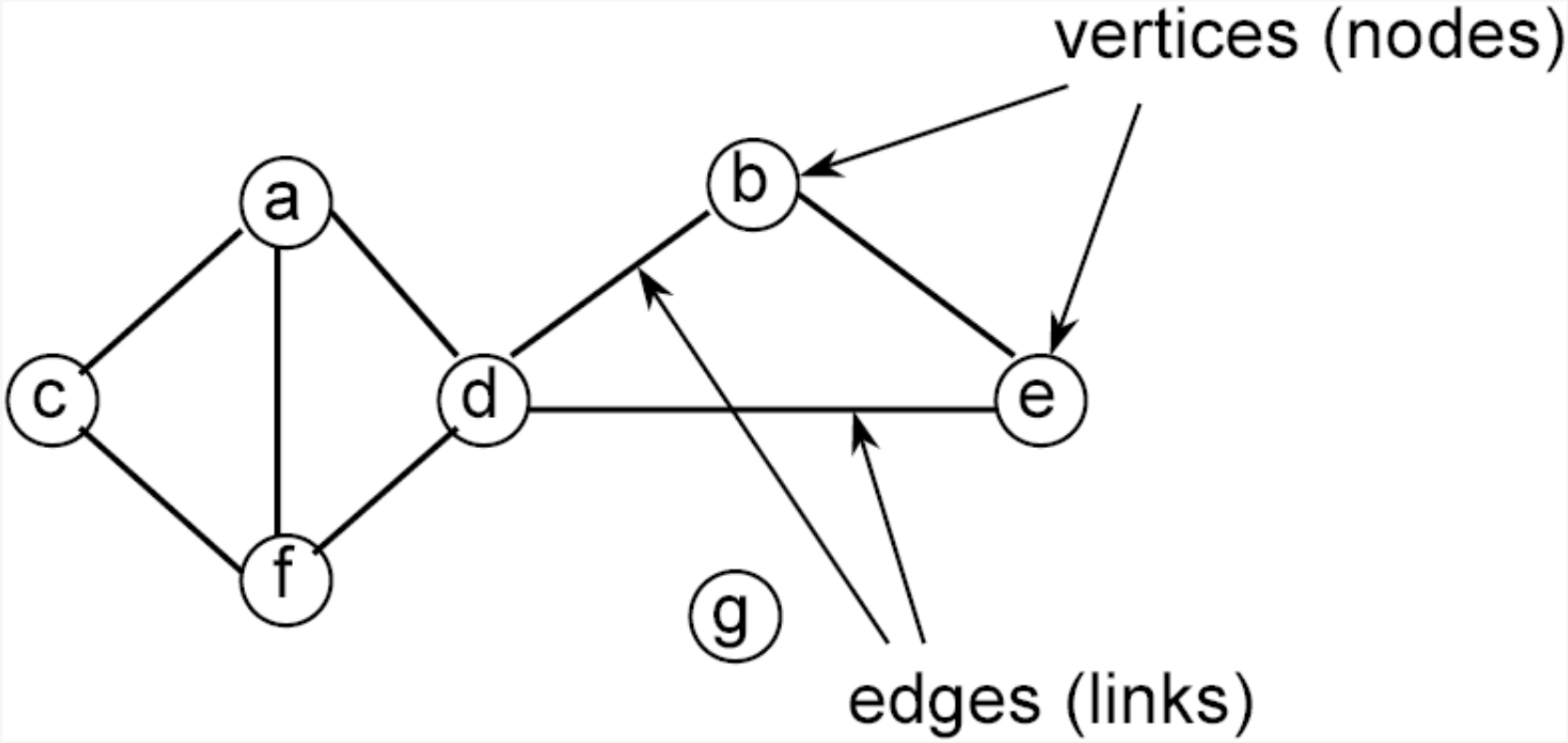
Graphs

Graphs allow the connection of any node to any other node

Think of maps, dependencies, relationships, flow networks, etc.

Algorithms to treat graphs vary, and often work by deriving a tree out of the graph

Graph components



Representing systems

Nodes can represent states (places, conditions, situations), or actions.

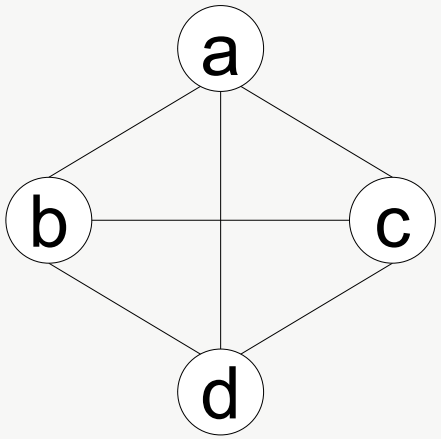
Edges represent links between the nodes: e.g., changes of state or precedence of actions.

Graphs are usually classified according to the information edges carry.

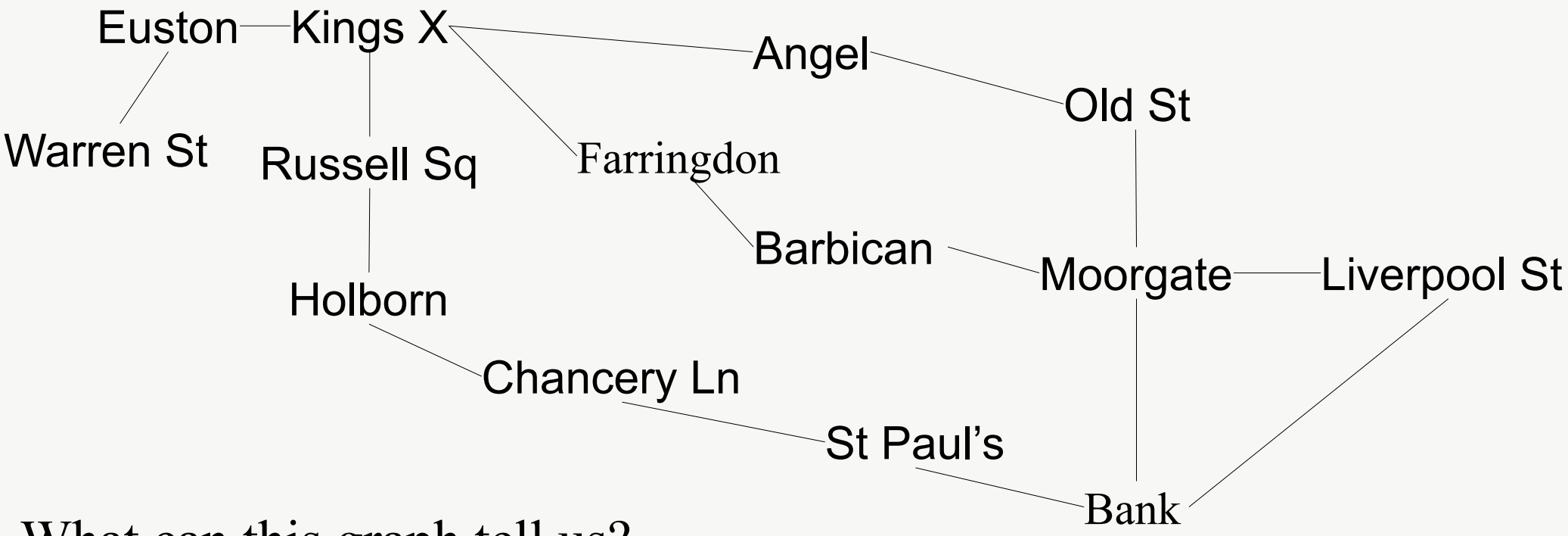
Simple graphs

- A simple graph consists of a set V of vertices, with at most one edge between each pair of vertices

- In simple graphs, edges represent a mutual relation between the nodes
- An edge is determined by the two nodes it joins



Example: the tube (simple graph)

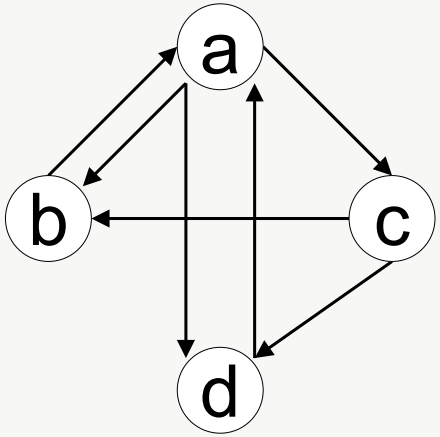


What can this graph tell us?

Directed graphs

A directed graph (or digraph) consists of:

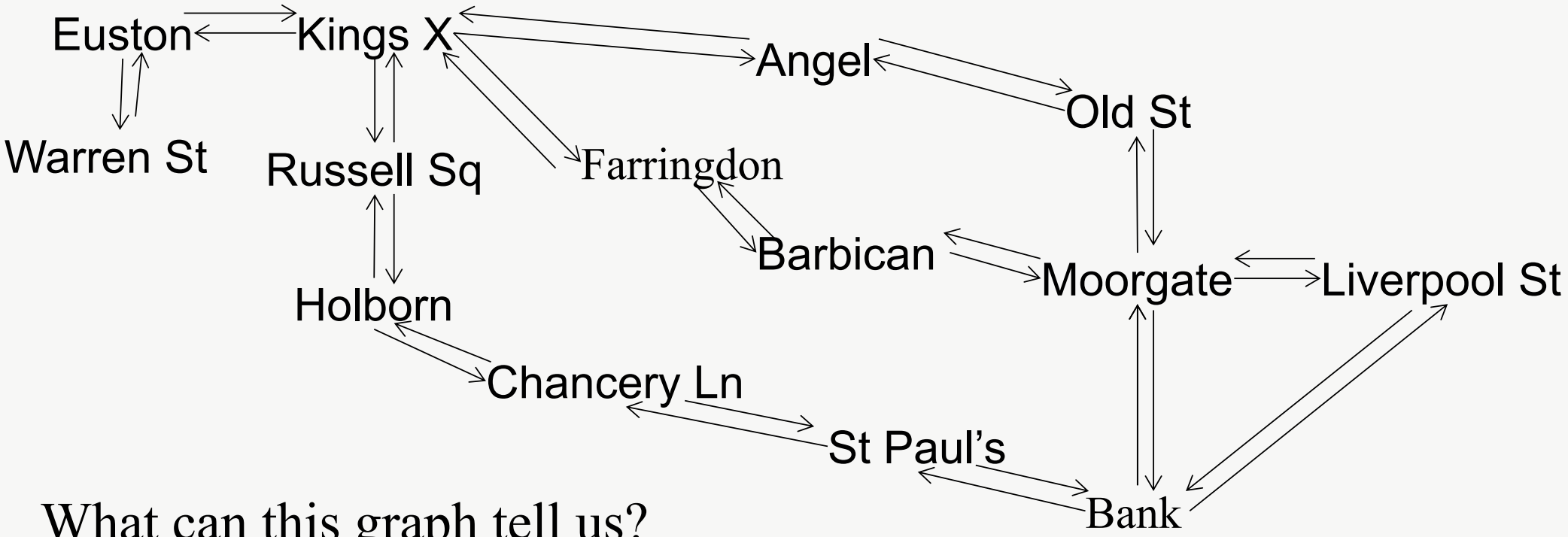
- a set V of vertices
- a set E of edges (in a fully connected digraph, E is equivalent to V x V)



Directed graphs (2)

- A graph can be viewed as representing a binary relation on V.
- An edge is determined by the precedent node, and the dependent node
- Simple graphs have a directed graph equivalent, but directed graphs may not have a simple graph equivalent

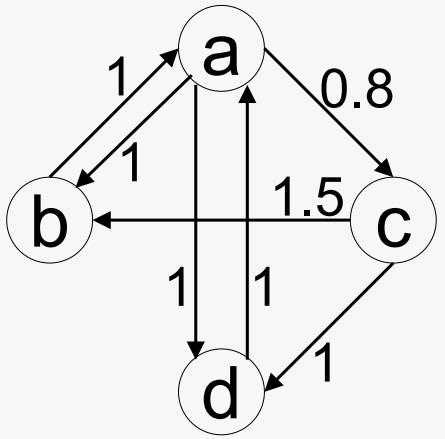
Example: the tube (directed graph)



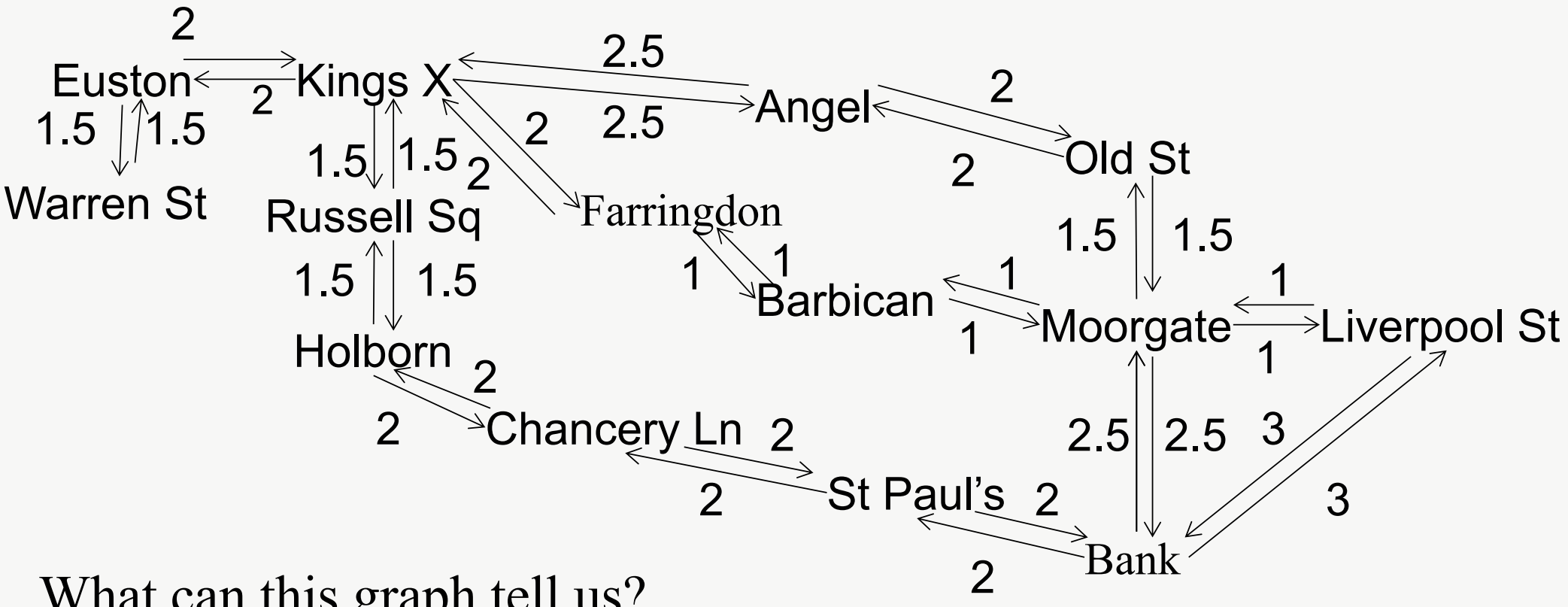
What can this graph tell us?

Weighted graphs

Edges contain additional information: a number representing time, cost or distance, for example.
This number is the “weight” of the edge .
If you are only considering “number of moves”, then the weight is 1 for each move (e.g., for each edge).
The weight (or cost) of a path is the sum of the weights of its edges.



Example: the tube (weighted graph)



What can this graph tell us?

Some definitions

- A vertex v_i is said to be **adjacent** to a vertex v_j if there is an edge from v_i to v_j .
- A **path** (or walk) from v_1 to v_n is a sequence of consecutive edges: v_1-->v_2-->v_3--> .... -->v_n .
- A **circuit** is a path from a vertex to itself, without repeating any edges.
- A **cycle** is a circuit in which no vertex is repeated.

Representing Graphs

Representing graphs

This far we have talked about what graphs represent

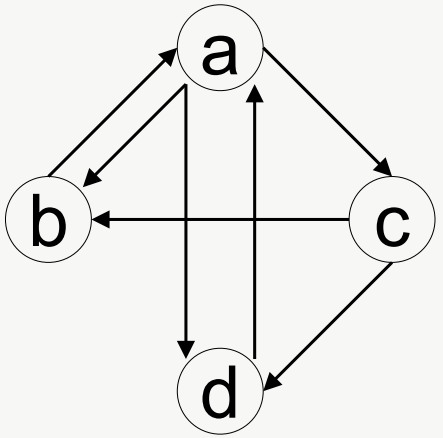
... but how can graphs be represented in a system?

Basically you need to represent the nodes and the links.

Adjacency matrix

An adjacency matrix is a $|V| \times |V|$ matrix a such that a_{ij} is 1 when there is an edge that goes from v_i to v_j.

Nodes are thus represented as rows and columns, and edges are represented as elements of the matrix.



nodes

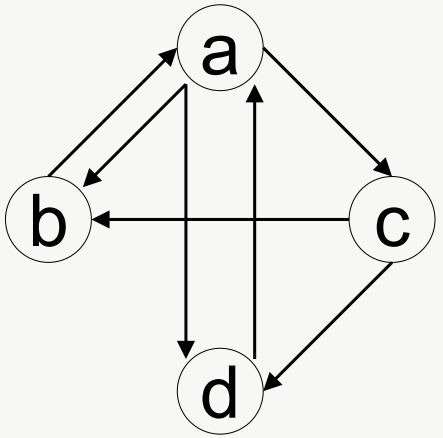
	a	b	c	d
a	0	1	1	1
b	1	0	0	0
c	0	1	0	1
d	1	0	0	0

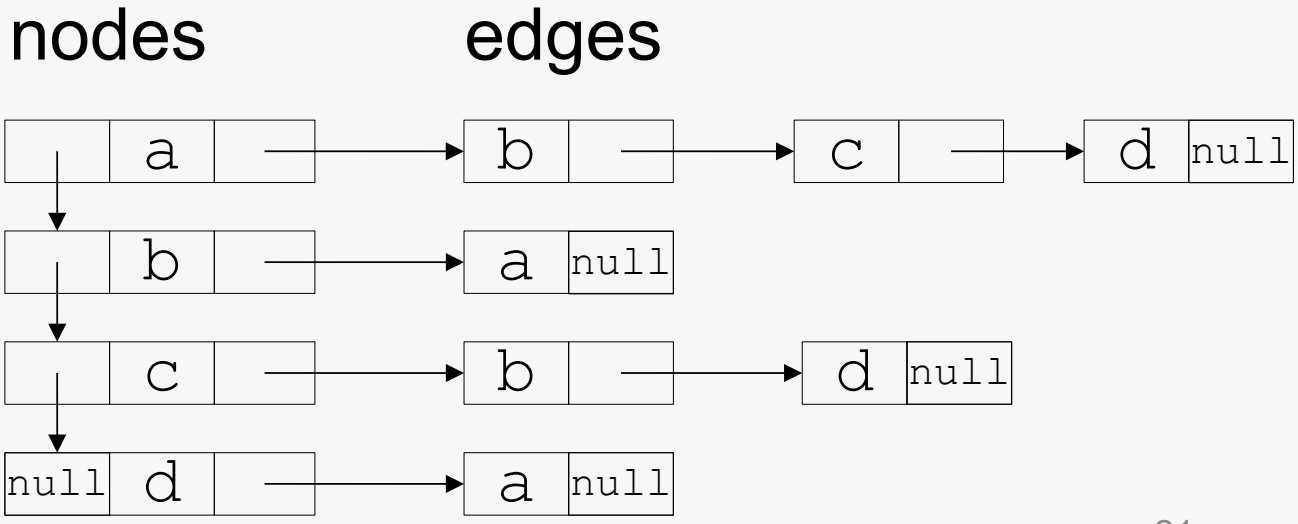
edges

Adjacency list

Adjacency lists show the dependants of a node in linked lists.

Nodes can be represented as elements in a linked list as well

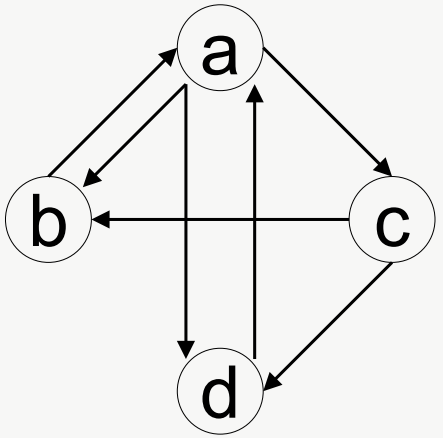


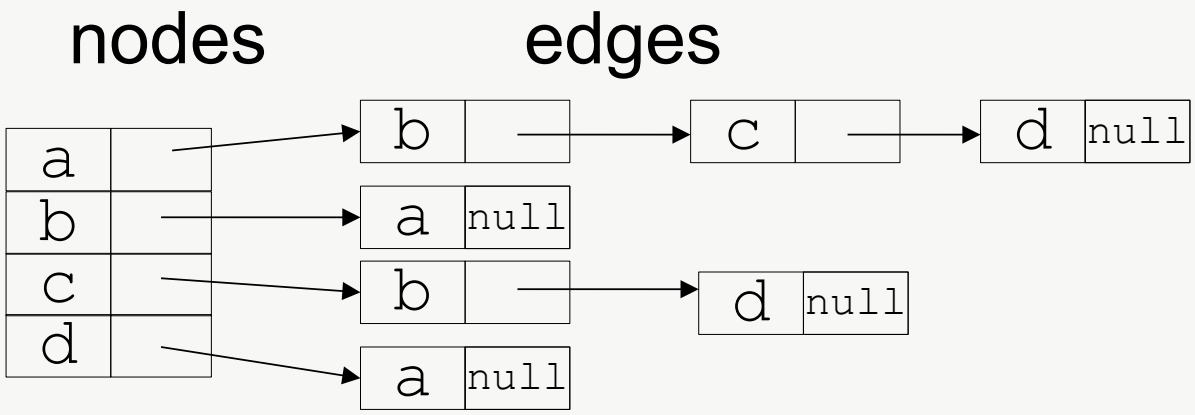


Adjacency list variant

Nodes can also be represented as elements in an array.

For sparse graphs an expandable array a good solution.





Graph traversals

Graph traversals

As with trees, graphs may be traversed depth-first or breadth-first

The algorithms are very similar to those for trees, except that encountered vertices need to be marked so that circuits do not lead to infinite traversing.

Depth-first traversal is used in other algorithms:

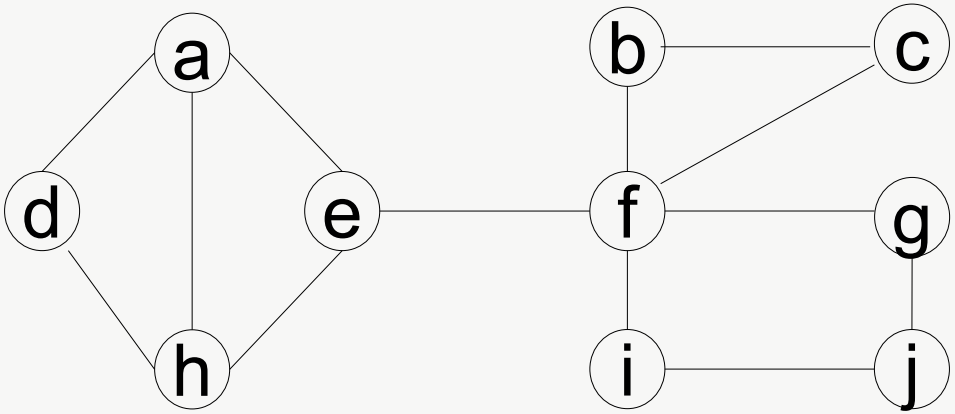
- **Topological sort**: order the vertices so no edge goes backwards
- **Strongly connected elements**: sets of vertices, each of which can be reached from any other

Depth-first traversal

Function dft(Vertex v):
visit v
FOR ALL vertices u adjacent to v
    IF u has not been visited THEN
        dft(u)

Function depthFirstSearch():
FOR ALL vertices v
    IF v has not been visited THEN
        dft(v)

Depth-first traversal

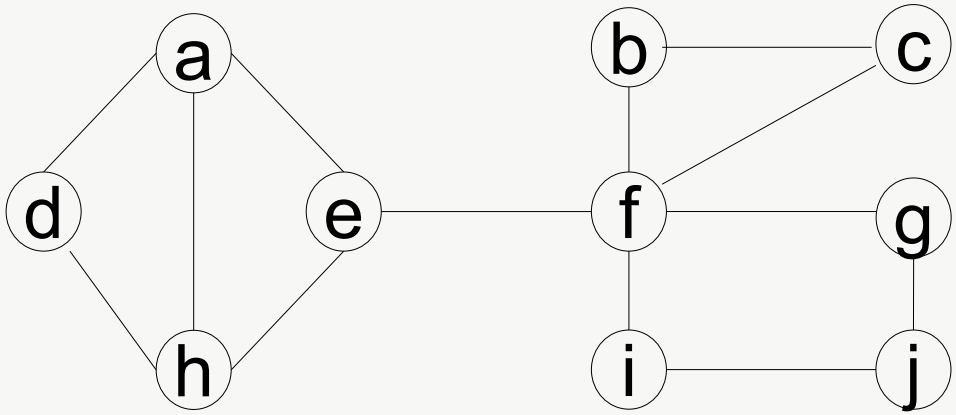


Breadth-first traversal

```
Function breadthFirstSearch():
create queue q
FOR ALL vertices v
    IF v has not been visited
        visit v
        enqueue v in q
        WHILE q is not empty
            u ← vertex dequeued from q
            FOR ALL vertices w adjacent to u
                IF w has not been visited
                    visit w
                    enqueue w in q
```

Note that only new vertices are added to the queue

Breadth-first traversal



Shortest Path Algorithms

Shortest path algorithms

all-to-all A simple algorithm takes $O(|V|^3)$ time, this can be improved a bit.

one-to-all There are several algorithms; we consider Dijkstra's, which assumes all weights are positive.

all-to-one Use a one-to-all algorithm on the reversed graph.

one-to-one No faster algorithm is known than using a one-to-all algorithm until the desired target is reached

Dijkstra's shortest path algorithm

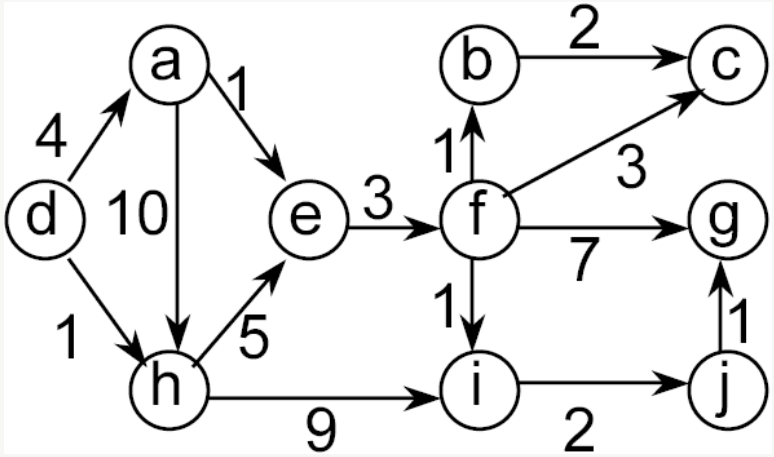
The algorithm maintains:

- A set **toBeChecked** of vertices whose shortest path is not yet known. Initially this set is V
- An array **currDist** giving the length of shortest path from start to every other vertex using the checked nodes. Initially 0 for start and - (or infinite) for every other vertex
- An array **predecessor** giving the previous vertex on the current shortest path, if any

Until toBeChecked is empty:

- extract from toBeChecked the closest vertex
- update adjacent vertices

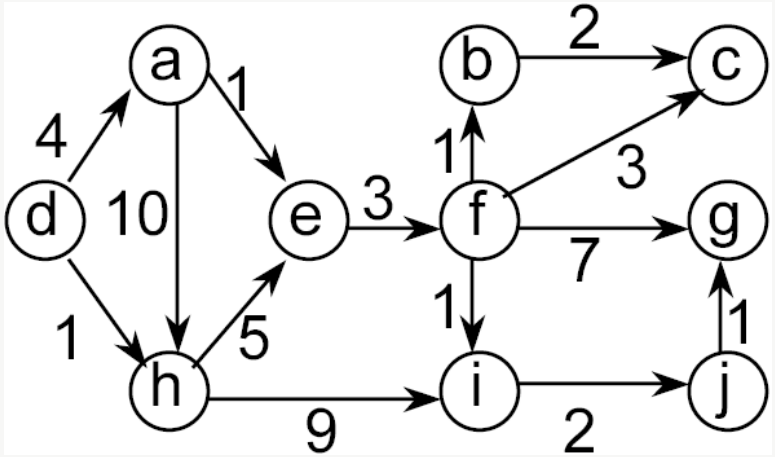
Example of Dijkstra's algorithm



Shortest distance from d

	a	b	c	d	e	f	g	h	i	j
toBeChecked	x	x	x	x	x	x	x	x	x	x
currDist	-	-	-	0	-	-	-	-	-	-
predecessor										

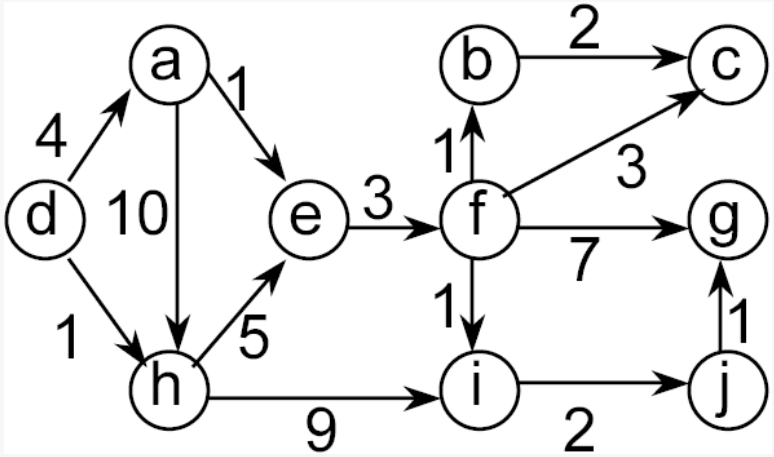
Example of Dijkstra's algorithm (2)



	a	b	c	d	e	f	g	h	i	j
toBeChecked	x	x	x		x	x	x	x	x	x
currDist	4	-	-	0	-	-	-	1	-	-
predecessor	d							d		

lowest unchecked goes next ↑

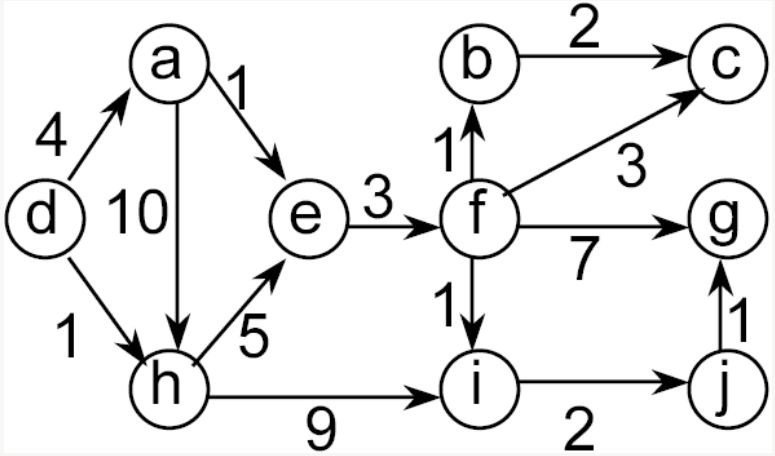
Example of Dijkstra's algorithm (3)



	a	b	c	d	e	f	g	h	i	j
toBeChecked										
currDist										
predecessor										

lowest unchecked goes next ↑

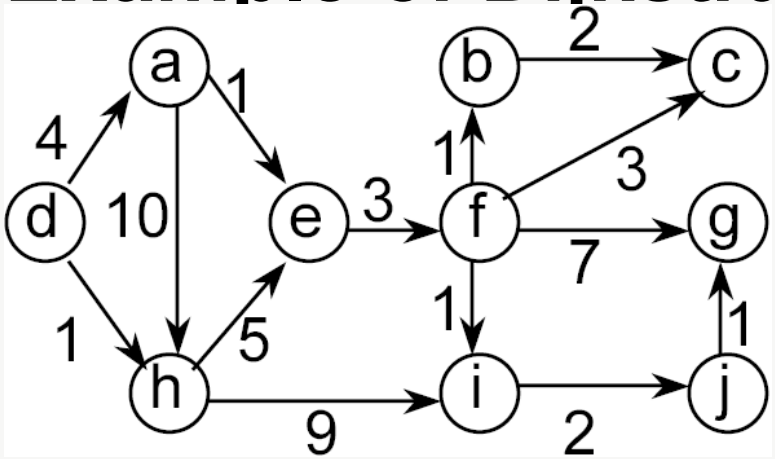
Example of Dijkstra's algorithm (4)



	a	b	c	d	e	f	g	h	i	j
toBeChecked										
currDist										
predecessor										

lowest unchecked goes next ↑

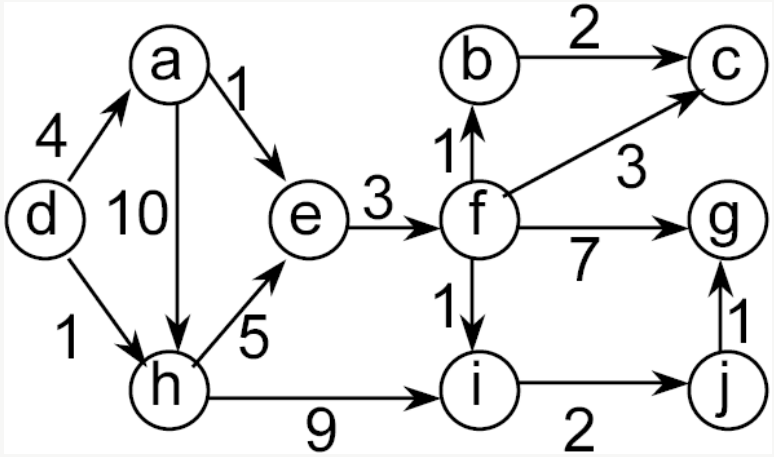
Example of Dijkstra's algorithm (5)



	a	b	c	d	e	f	g	h	i	j
toBeChecked										
currDist										
predecessor										

↑ lowest unchecked goes next

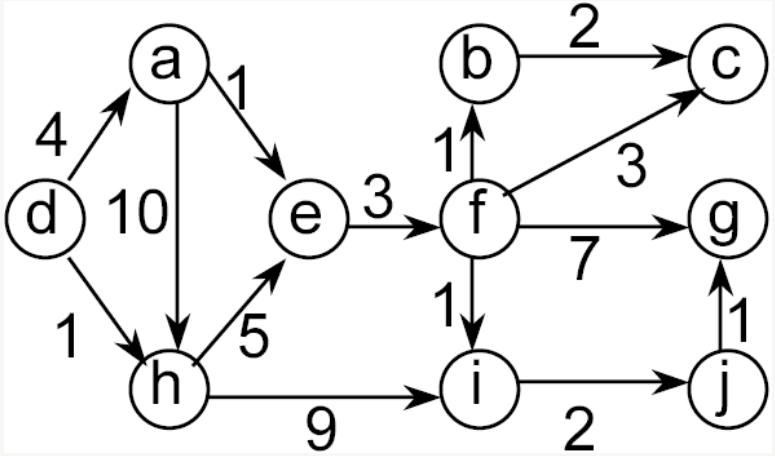
Example of Dijkstra's algorithm (6)



	a	b	c	d	e	f	g	h	i	j
toBeChecked										
currDist										
predecessor										

lowest unchecked goes next ↑

Example of Dijkstra's algorithm (skipped to the end)



	a	b	c	d	e	f	g	h	i	j
toBeChecked										
currDist	4	9	11	0	5	8	12	1	9	11
predecessor	d	f	f		a	e	j	d	f	i

Dijkstra's algorithm

```
FOR all vertices v
        currDist[v] <- infinity;
currDist[start] <- 0;
toBeChecked <- V;
WHILE toBeChecked is not empty
        v <- a vertex in toBeChecked with minimal currDist(v);
        remove v from toBeChecked;
        FOR all vertices u adjacent to v and in toBeChecked
                IF currDist[u] > currDist[v] + weight(vu)
                        currDist[u] <- currDist[v] + weight(vu);
                        predecessor[u] <- v;
```

Analysis of Dijkstra's algorithm

With adjacency list representation:

- The outer while loop is executed O(|V|) times, and on each iteration, extracting vertices takes O(|V|) time
- The inner for loop is executed a total of O(|E|) times, performing O(1) work each time
- Giving a total of $O(|V|^2)$ time (because $|E| <= |V|^2$)

Graph Problems

Graph problems

Polynomial time algorithms exist for the following problems:

- all-to-all shortest paths
- topological sort order the vertices so no edge goes backwards
- strongly connected components sets of vertices, each of which can be reached from any other
- minimal spanning tree: find a minimal cost tree that covers all vertices (Kruskal algorithm)
- network flow (Dinic algorithm)

Hard graph problems

Recall **NP-complete** problems: the best known algorithms are exponential, and if any one has a polynomial algorithm, it can be adapted to all of them.

Hence it is conjectured that no such algorithm exists.

NP-complete graph problems include:

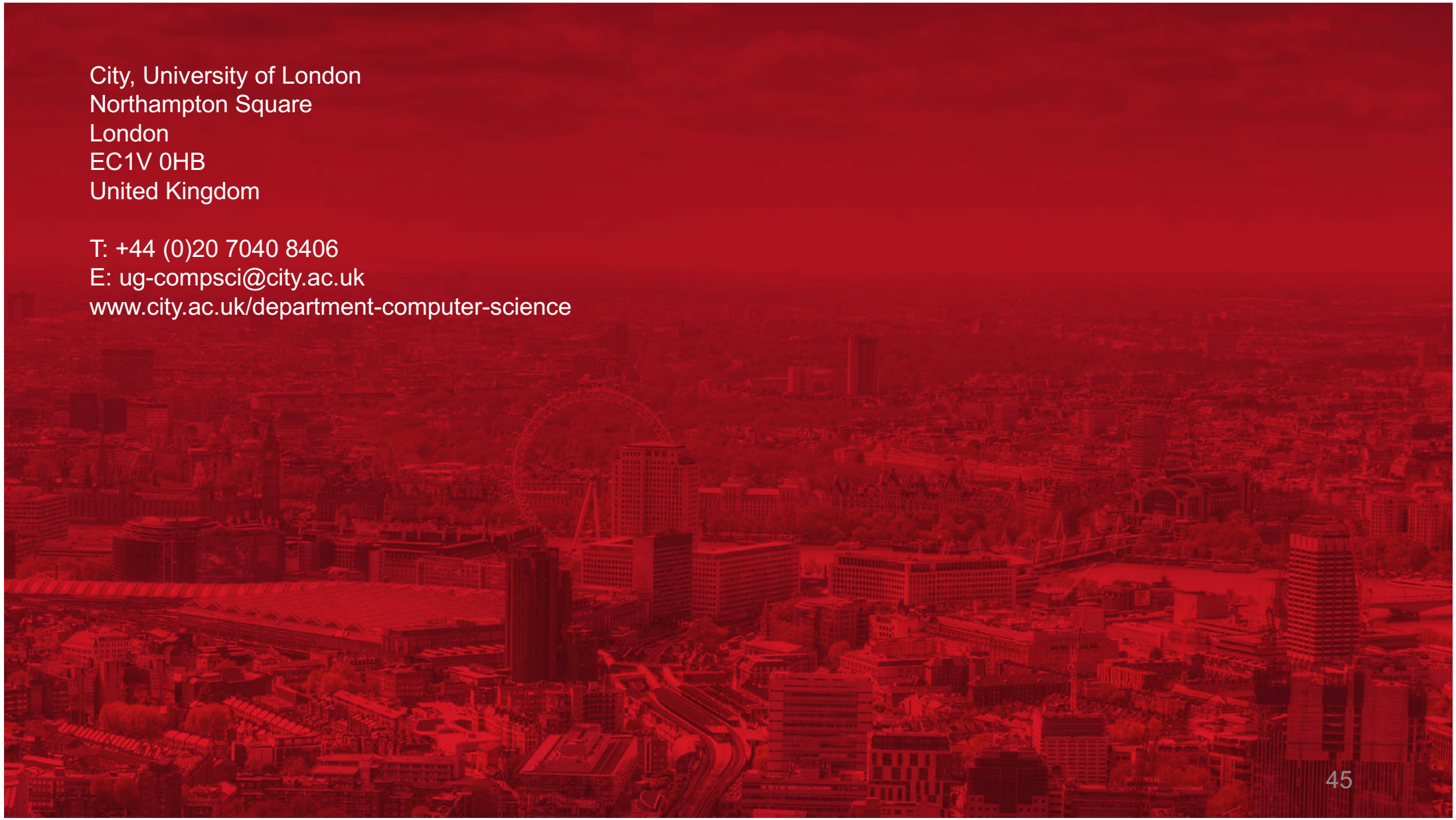
- travelling salesman: Find the shortest circuit that includes all vertices
- longest cycle
- largest clique

Other problems are unclassified, e.g., testing whether two graphs are equivalent (isomorphic).

Reading

- Weiss: Sections 13.1-13.4
- Drozdek: Sections 8.1, 8.2 and 8.3

Next session: Revision



City, University of London
Northampton Square
London
EC1V 0HB
United Kingdom

T: +44 (0)20 7040 8406
E: ug-compsci@city.ac.uk
www.city.ac.uk/department-computer-science