Data Structures and Algorithms – COMS21103

2014/2015

Representing and Exploring Graphs

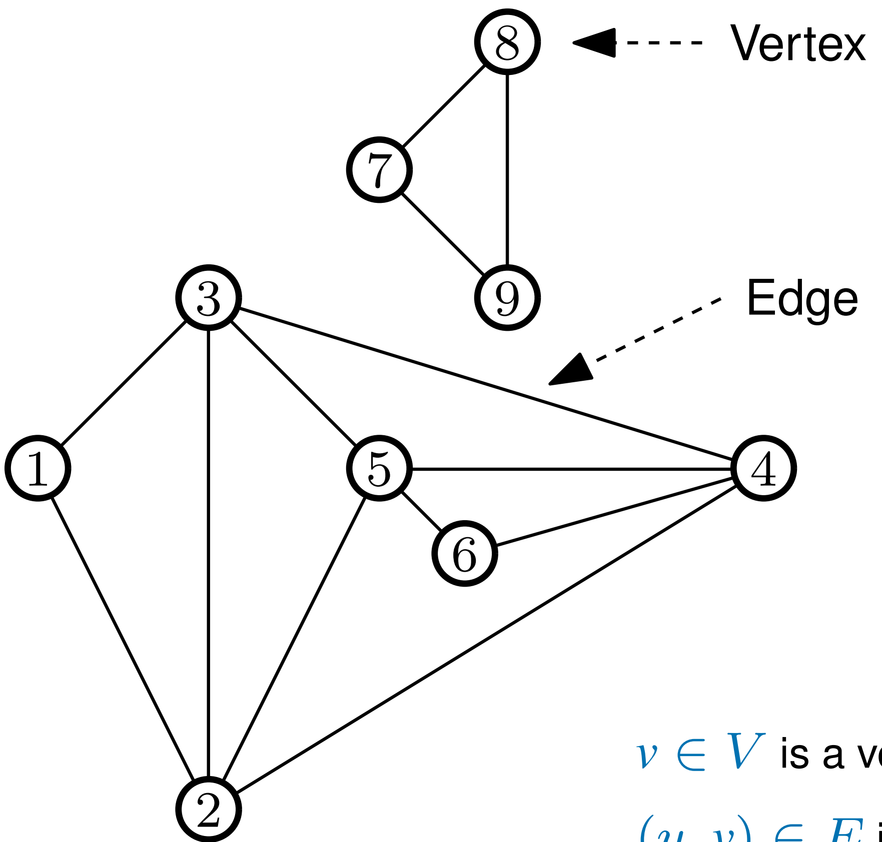
Depth First Search and Breadth First Search

Benjamin Sach

University of BRISTOL

Graph notation recap

G is a Graph,



V is the set of vertices

$|V|$ *is the number of vertices*

E is the set of edges

$|E|$ *is the number of edges*

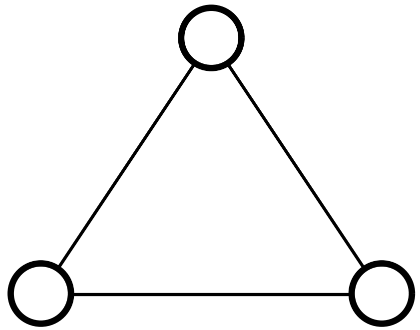
$v \in V$ is a vertex

$(u, v) \in E$ is an edge from u to v

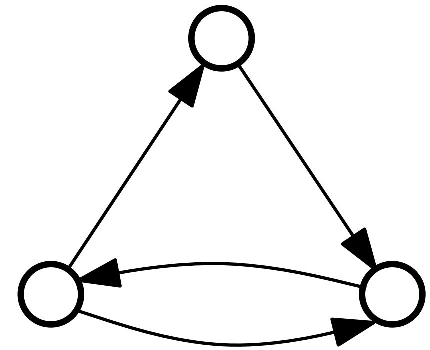
This lecture focuses on exploring **undirected** and **unweighted** graphs

though everything discussed today will work for directed, unweighted graphs too

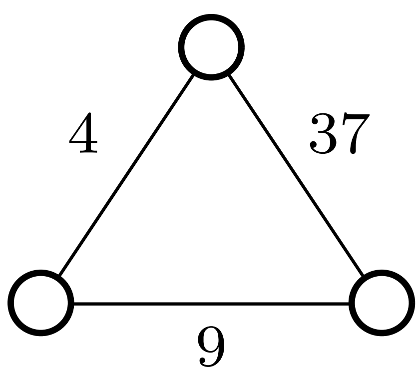
Graph notation recap



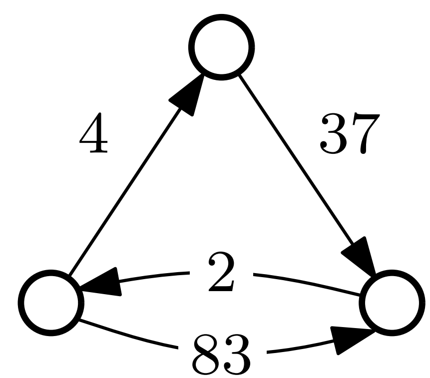
an **undirected** and **unweighted** graph



an **directed** and **unweighted** graph



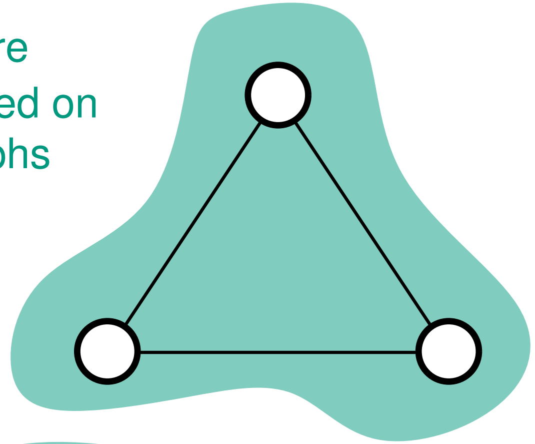
an **undirected** and **weighted** graph



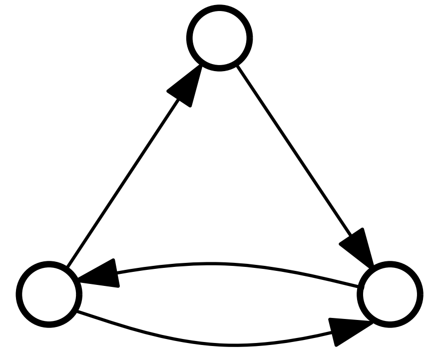
a **directed** and **weighted** graph

Graph notation recap

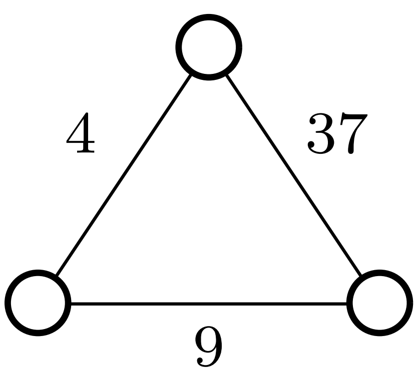
Today we're focused on these graphs



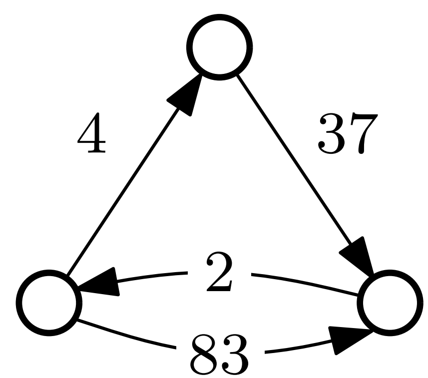
an **undirected** and **unweighted** graph



an **directed** and **unweighted** graph



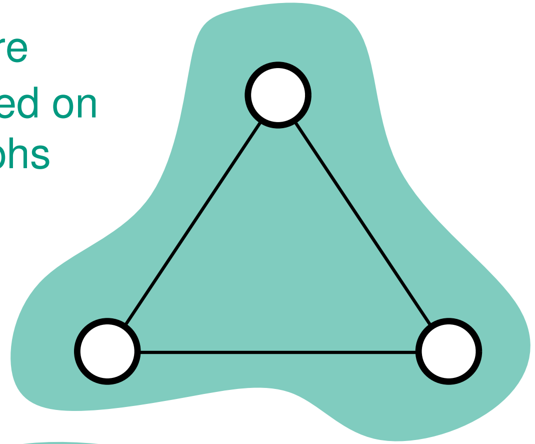
an **undirected** and **weighted** graph



a **directed** and **weighted** graph

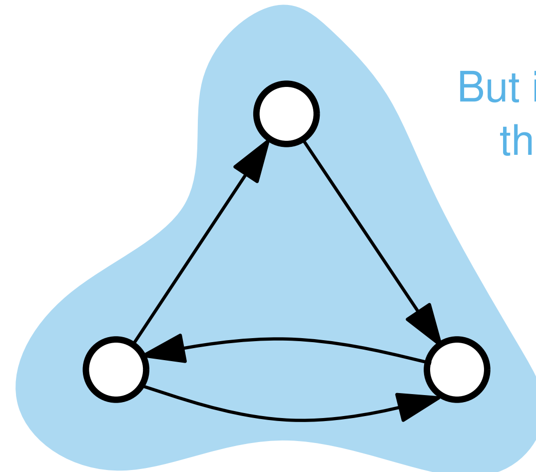
Graph notation recap

Today we're focused on these graphs

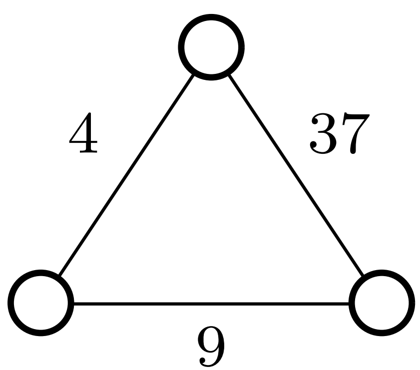


an **undirected** and **unweighted** graph

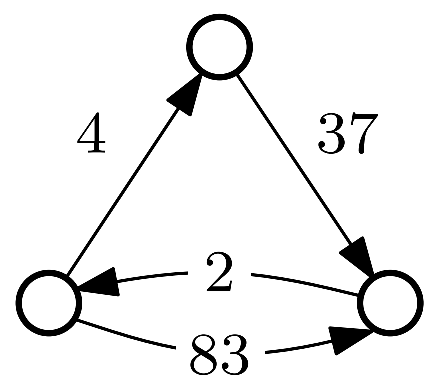
But it all works for these graphs too



an **directed** and **unweighted** graph



an **undirected** and **weighted** graph

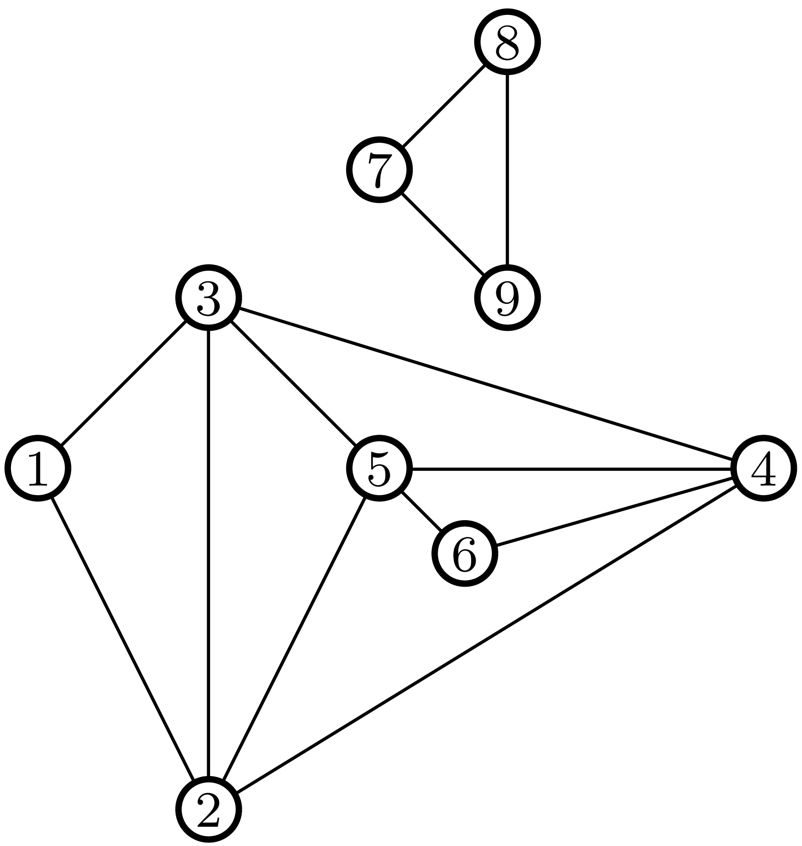


a **directed** and **weighted** graph

V is the set of vertices
$|V|$ *is the number of vertices*

Graph representations

E is the set of edges
$|E|$ *is the number of edges*



TO

FROM

$$\begin{array}{c|ccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
3 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
5 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\hline
\end{array}$$

$|V|$

$|V|$

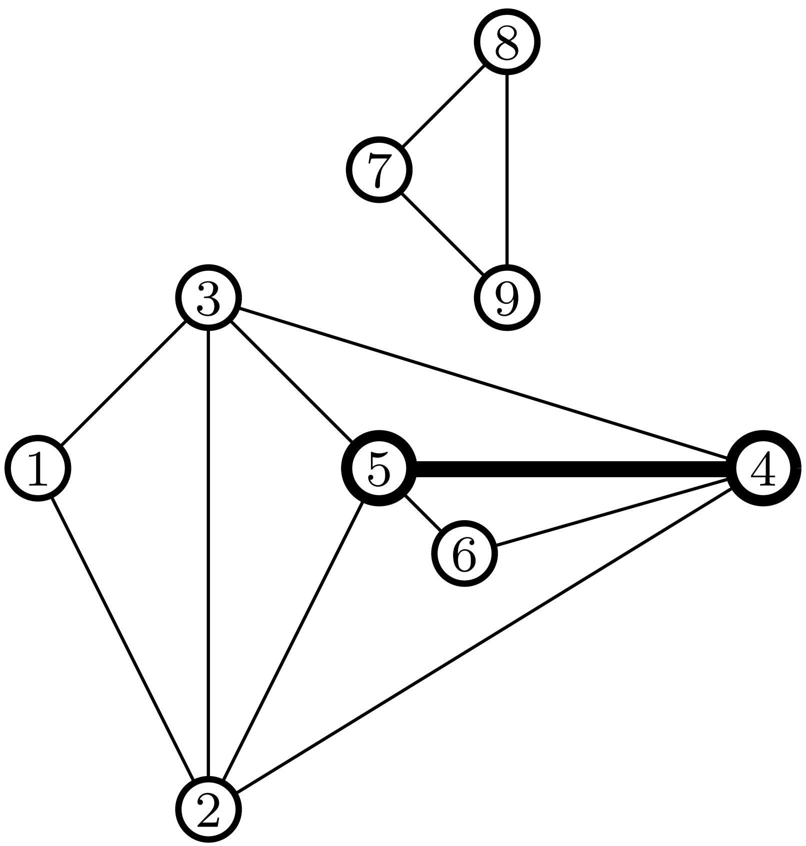
Adjacency Matrix

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

V is the set of vertices
$|V|$ *is the number of vertices*

Graph representations

E is the set of edges
$|E|$ *is the number of edges*



TO

FROM	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	0	0
2	1	0	1	1	1	0	0	0	0
3	1	1	0	1	1	0	0	0	0
4	0	1	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0	0
6	0	0	0	1	1	0	0	0	0
7	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	1	0	1
9	0	0	0	0	0	0	1	1	0

$|V|$ $|V|$

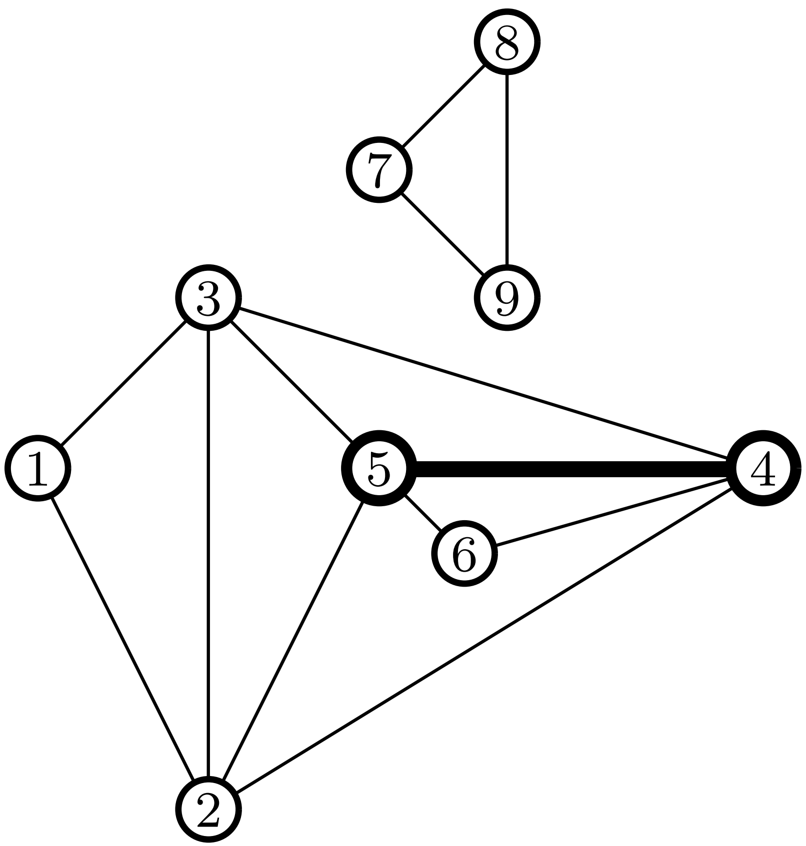
Adjacency Matrix

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

V is the set of vertices
$|V|$ is the number of vertices

Graph representations

E is the set of edges
$|E|$ is the number of edges



TO

FROM	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	0	0
2	1	0	1	1	1	0	0	0	0
3	1	1	0	1	1	0	0	0	0
4	0	1	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0	0
6	0	0	0	1	1	0	0	0	0
7	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	1	0	1
9	0	0	0	0	0	0	1	1	0

$|V|$ $|V|$

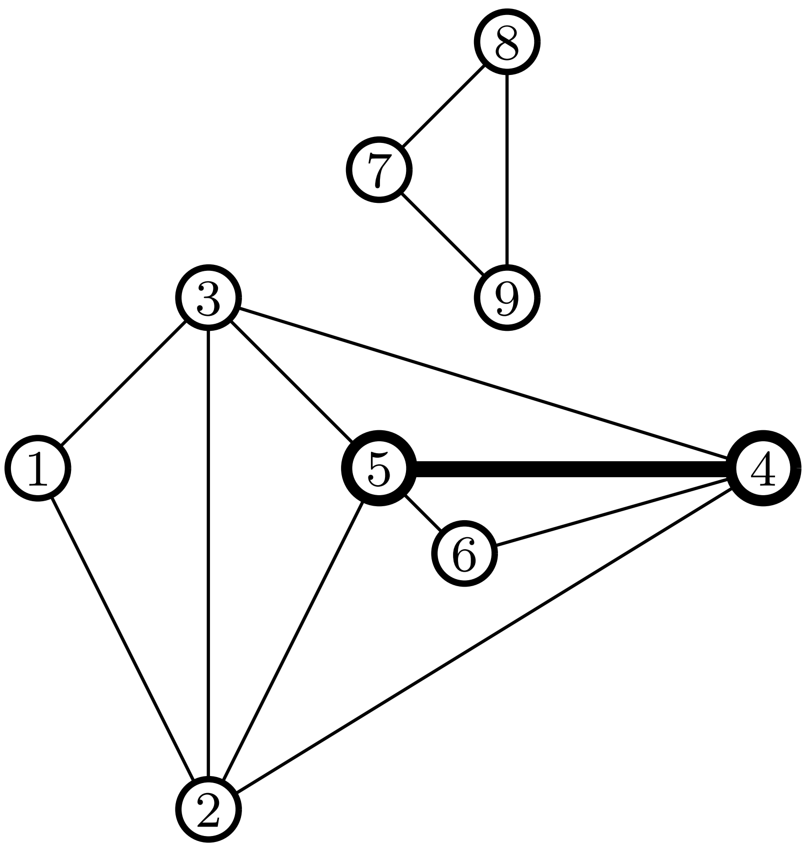
Adjacency Matrix

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

V is the set of vertices
$|V|$ *is the number of vertices*

Graph representations

E is the set of edges
$|E|$ *is the number of edges*



TO

FROM	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	0	0
2	1	0	1	1	1	0	0	0	0
3	1	1	0	1	1	0	0	0	0
4	0	1	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0	0
6	0	0	0	1	1	0	0	0	0
7	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	1	0	1
9	0	0	0	0	0	0	1	1	0

$|V|$ $|V|$

Adjacency Matrix

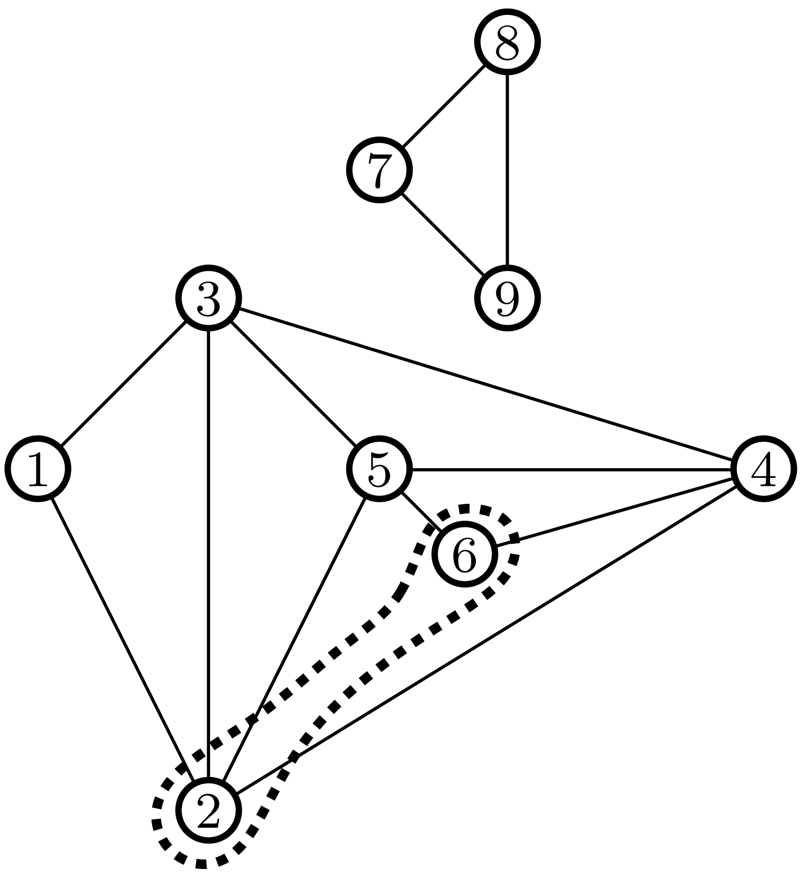
We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

University of BRISTOL

V is the set of vertices
$|V|$ is the number of vertices

Graph representations

E is the set of edges
$|E|$ is the number of edges



TO

FROM	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	0	0
2	1	0	1	1	1	0	0	0	0
3	1	1	0	1	1	0	0	0	0
4	0	1	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0	0
6	0	0	0	1	1	0	0	0	0
7	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	1	0	1
9	0	0	0	0	0	0	1	1	0

$|V|$

$|V|$

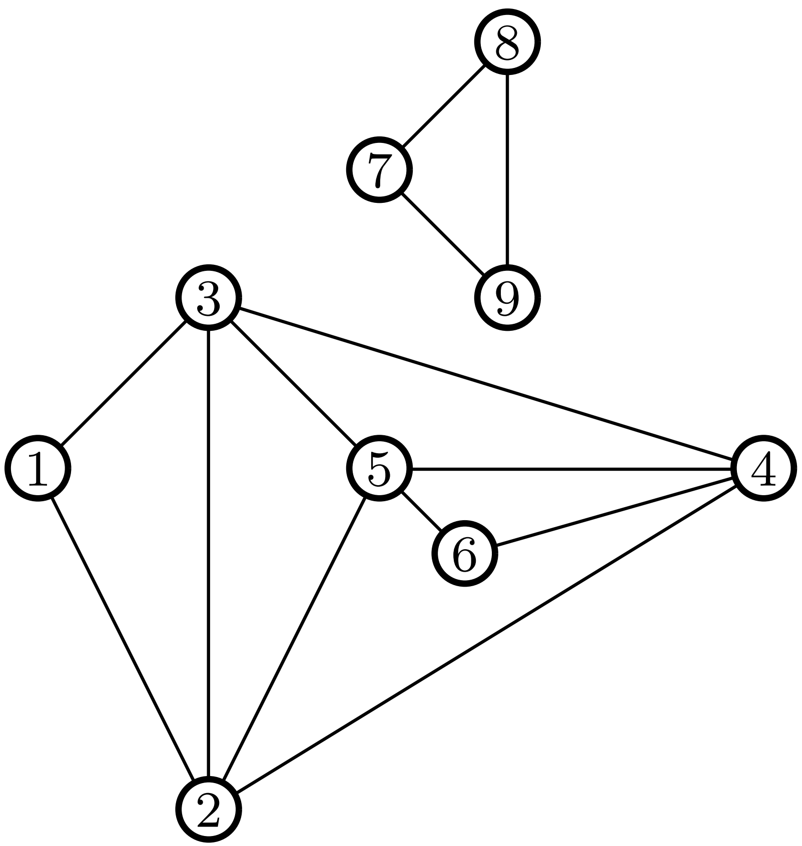
Adjacency Matrix

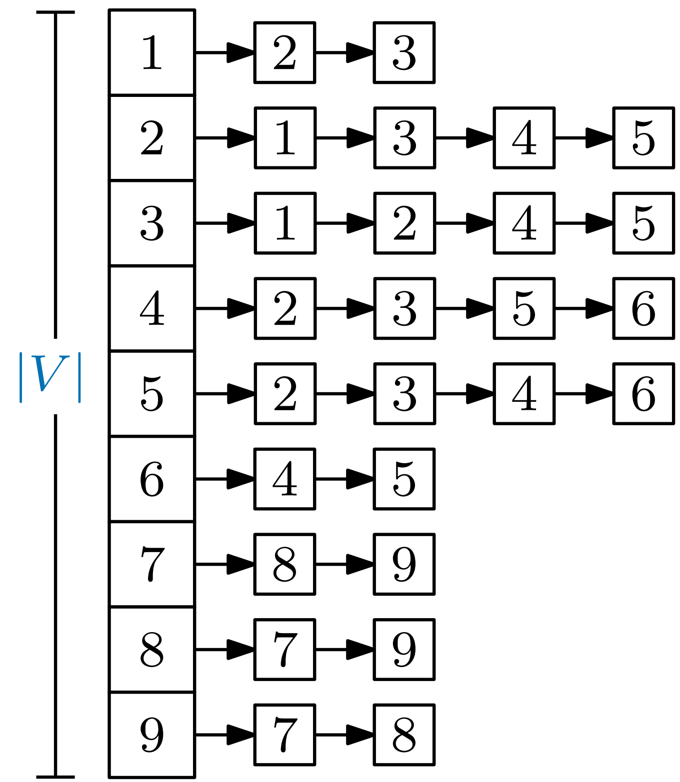
We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

V is the set of vertices
$|V|$ is the number of vertices

Graph representations

E is the set of edges
$|E|$ is the number of edges





Adjacency List

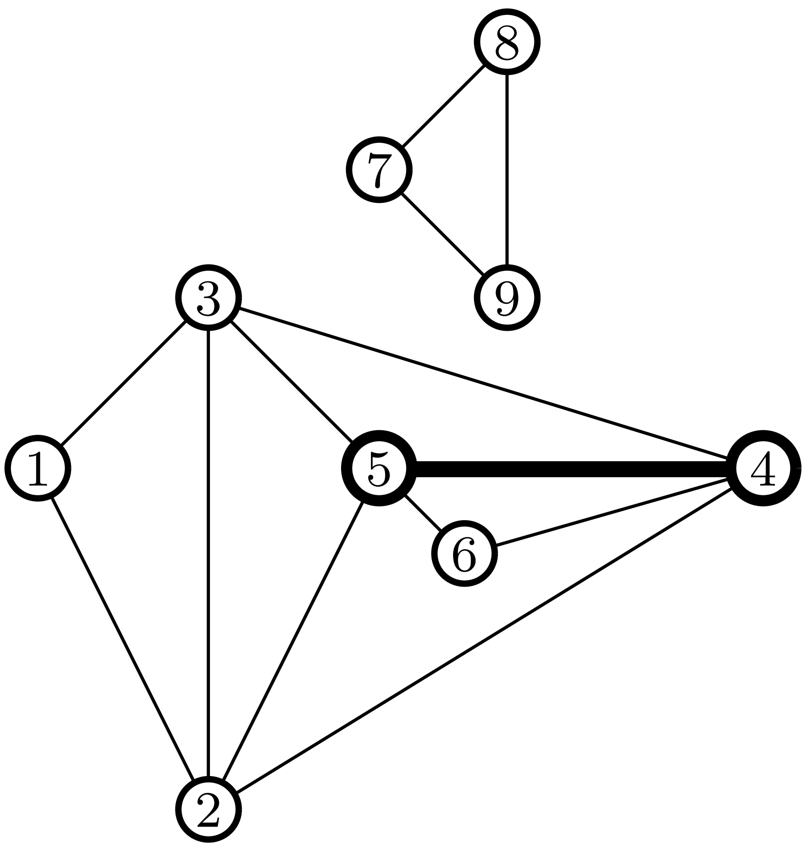
(using linked lists)

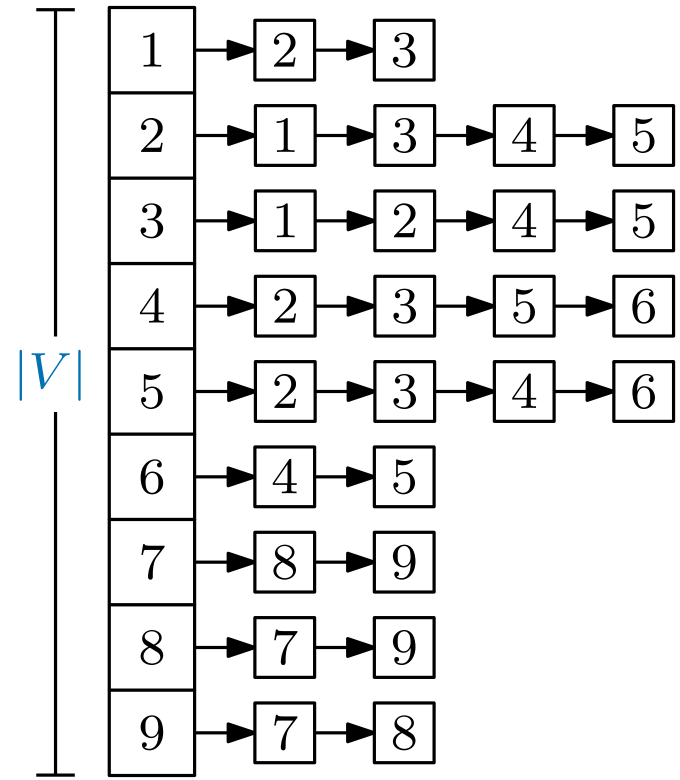
We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

V is the set of vertices
$|V|$ is the number of vertices

Graph representations

E is the set of edges
$|E|$ is the number of edges





Adjacency List

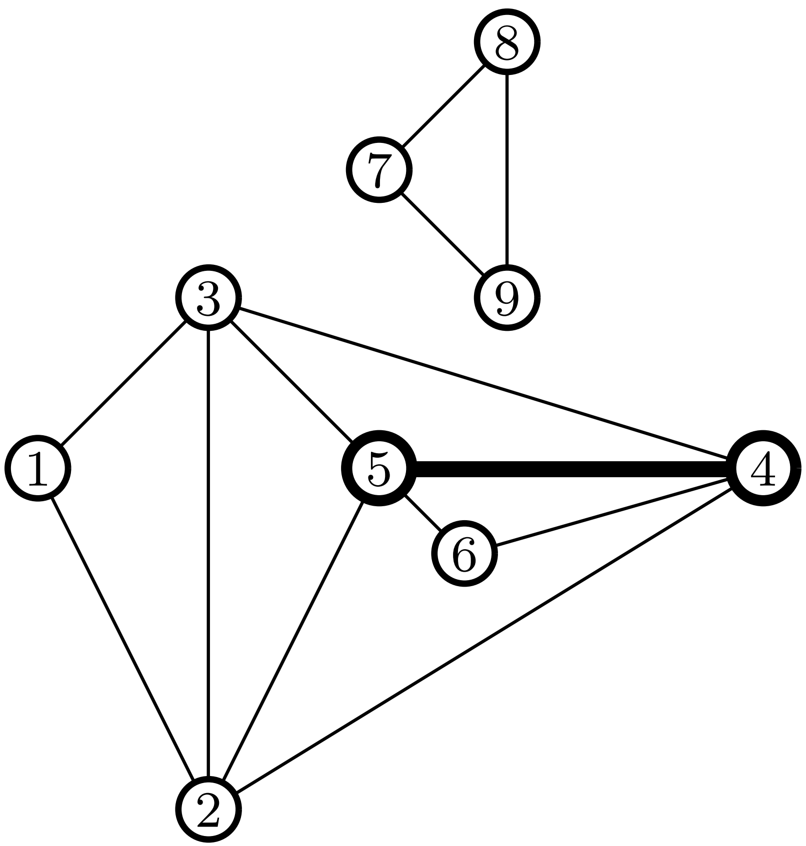
(using linked lists)

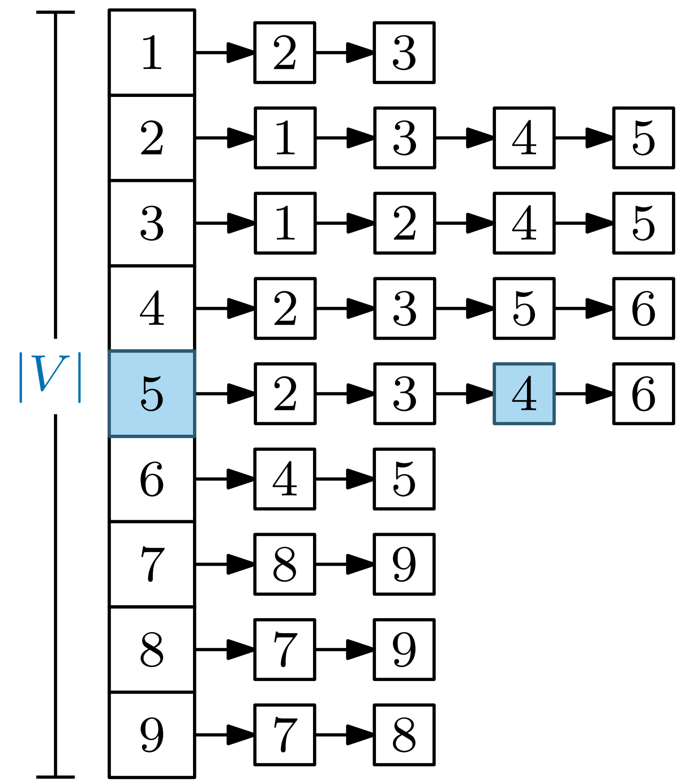
We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

V is the set of vertices
$|V|$ is the number of vertices

Graph representations

E is the set of edges
$|E|$ is the number of edges



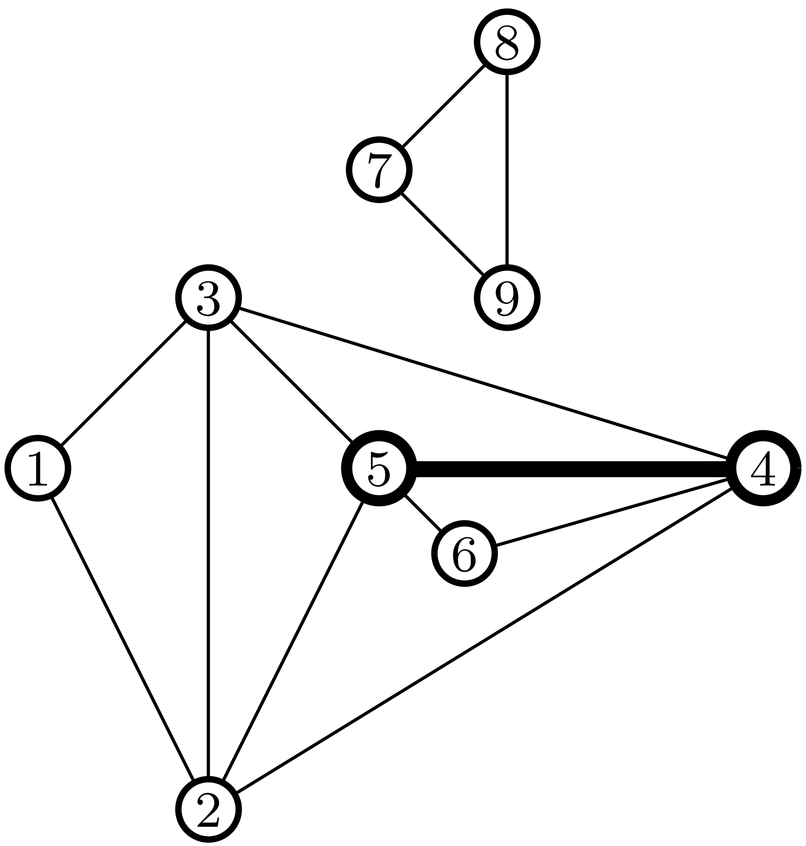


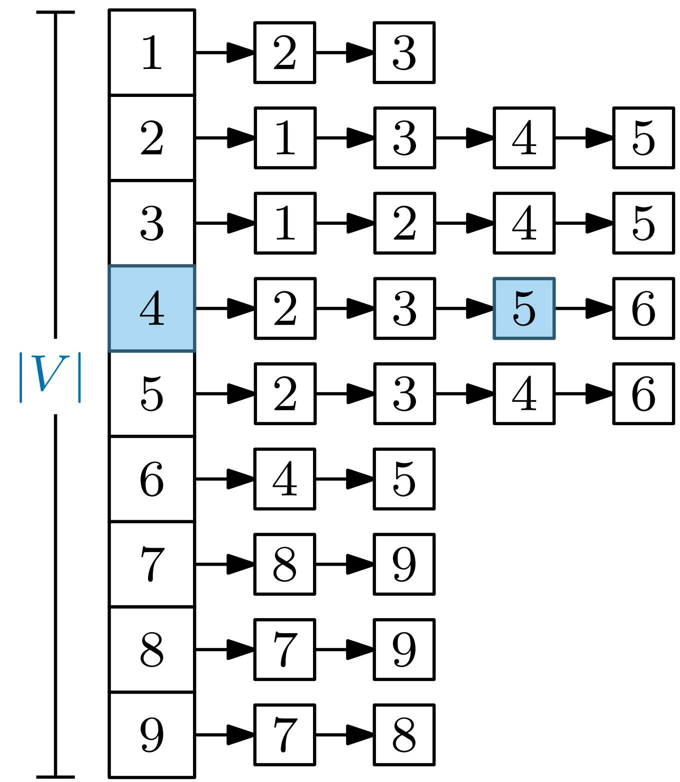
Adjacency List

(using linked lists)

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

Graph representations





Adjacency List

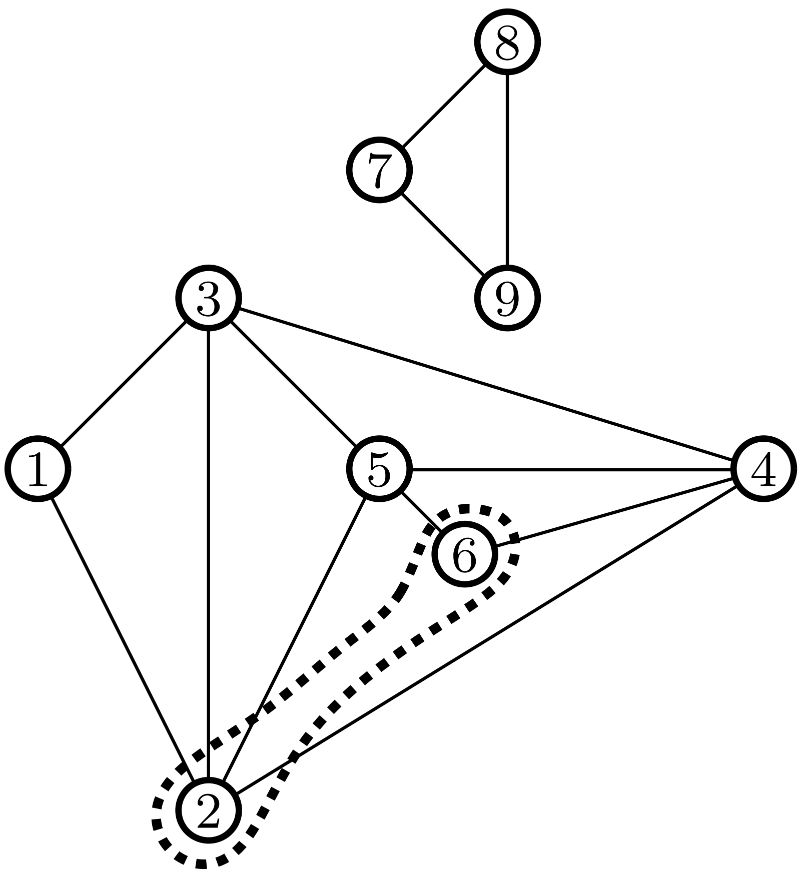
(using linked lists)

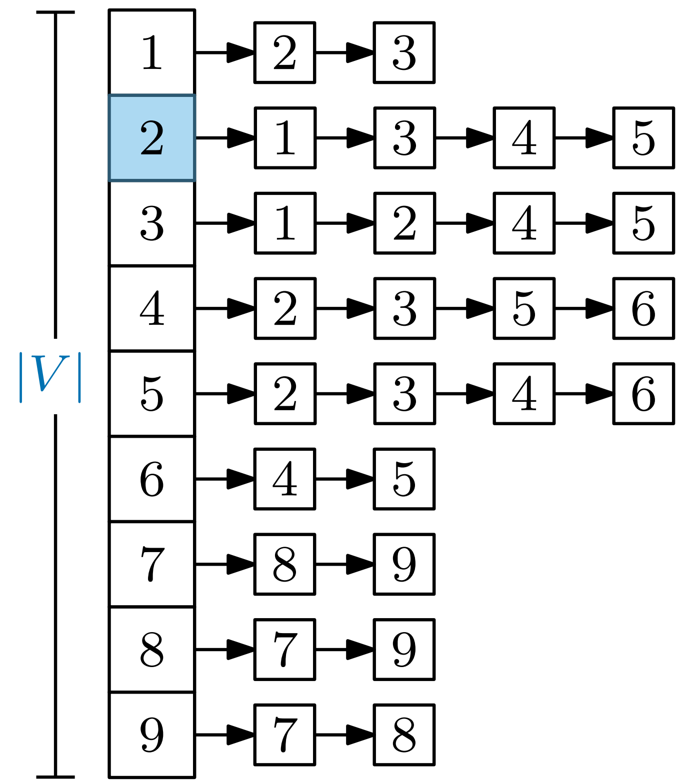
We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

V is the set of vertices
$|V|$ *is the number of vertices*

Graph representations

E is the set of edges
$|E|$ *is the number of edges*





Adjacency List

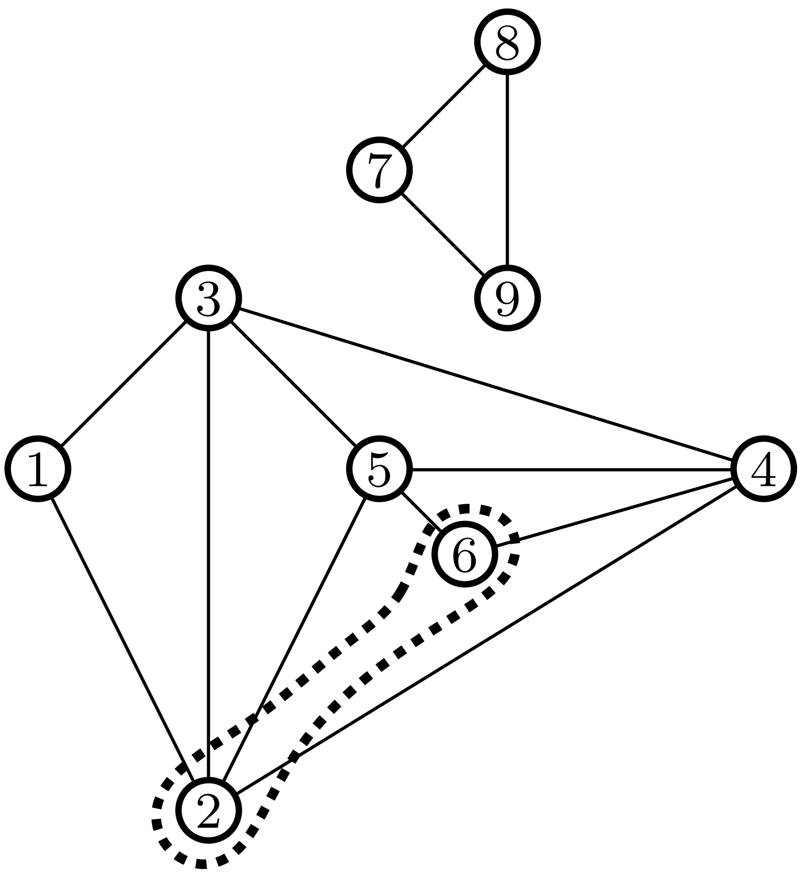
(using linked lists)

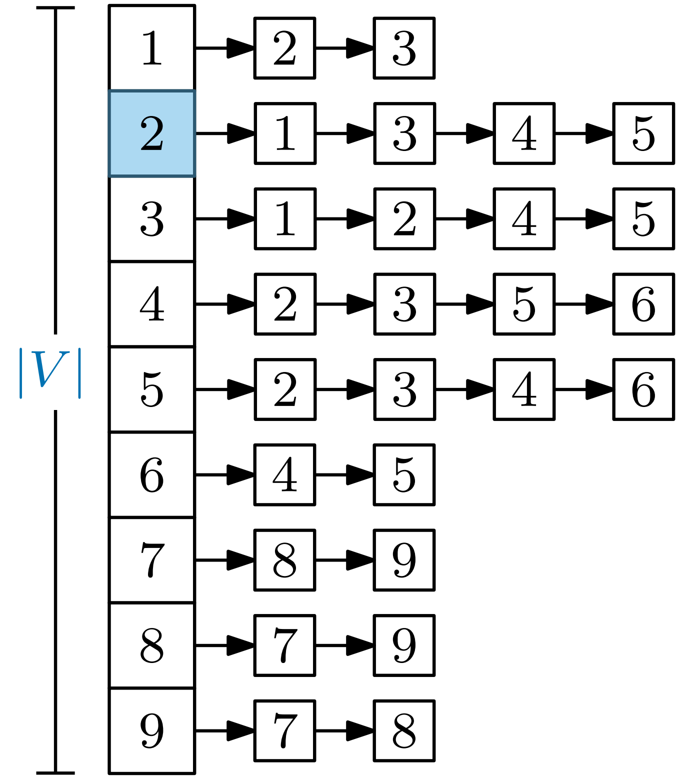
We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

V is the set of vertices
$|V|$ *is the number of vertices*

Graph representations

E is the set of edges
$|E|$ *is the number of edges*





Adjacency List

(using linked lists)

We will in general assume that the vertices are numbered $1, 2, 3 \ldots |V|$

both representations are symmetric because the graphs are undirected

V is the set of vertices
$|V|$ *is the number of vertices*

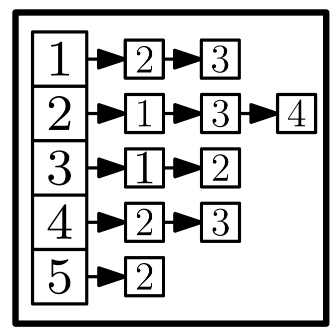
Basic operations

E is the set of edges
$|E|$ *is the number of edges*

	Adjacency Matrix	Adjacency List *(using linked lists)*
Space		
Is there an edge from vertex u to v?		
List all the edges leaving $u \in V$		

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

Adjacency Matrix



Adjacency List
(using linked lists)

V is the set of vertices
$|V|$ *is the number of vertices*

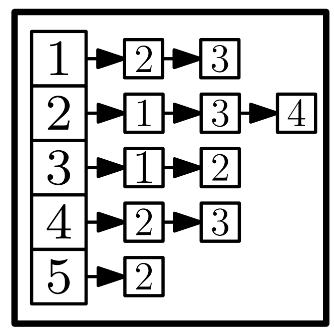
Basic operations

E is the set of edges
$|E|$ *is the number of edges*

	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$
Is there an edge from vertex u to v?		
List all the edges leaving $u \in V$		

$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 1 & 1 & 0 & 1 & 1 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 0 & 1 & 1 & 1 & 0
\end{array}
$$

Adjacency Matrix



Adjacency List
(using linked lists)

V is the set of vertices
$|V|$ *is the number of vertices*

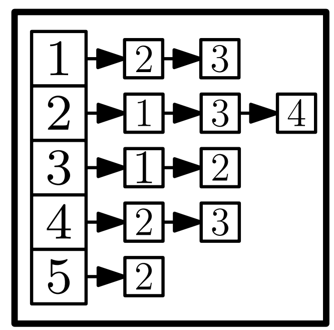
Basic operations

E is the set of edges
$|E|$ *is the number of edges*

	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	
List all the edges leaving $u \in V$		

$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 1 & 1 & 0 & 1 & 1 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 0 & 1 & 1 & 1 & 0 \\
\hline
\end{array}
$$

Adjacency Matrix



Adjacency List
(using linked lists)

V is the set of vertices
$|V|$ *is the number of vertices*

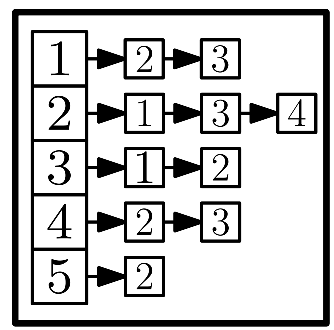
Basic operations

E is the set of edges
$|E|$ *is the number of edges*

	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(V)$ time
List all the edges leaving $u \in V$		

$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 1 & 1 & 0 & 1 & 1 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 0 & 1 & 1 & 1 & 0 \\
\hline
\end{array}
$$

Adjacency Matrix



Adjacency List
(using linked lists)

V is the set of vertices
$|V|$ *is the number of vertices*

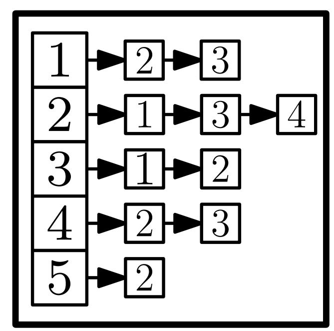
Basic operations

E is the set of edges
$|E|$ *is the number of edges*

	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time
List all the edges leaving $u \in V$		

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

Adjacency Matrix



Adjacency List
(using linked lists)

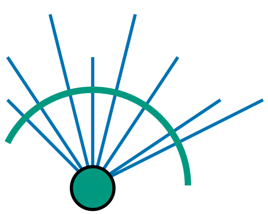
V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time
List all the edges leaving $u \in V$		

$\deg(u)$ *(the degree of u),*
is the number of edges leaving u



$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 & 1 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 1 & 1 & 1 & 0 \\ \hline \end{array}$$

Adjacency Matrix

$1 \to 2 \to 3$
$2 \to 1 \to 3 \to 4$
$3 \to 1 \to 2$
$4 \to 2 \to 3$
$5 \to 2$

Adjacency List
(using linked lists)

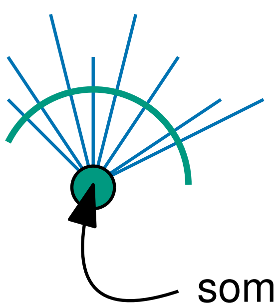
V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time
List all the edges leaving $u \in V$		

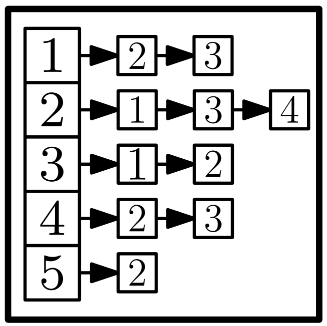
$\deg(u)$ *(the degree of u),*
is the number of edges leaving u



some vertex u

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

Adjacency Matrix



Adjacency List
(using linked lists)

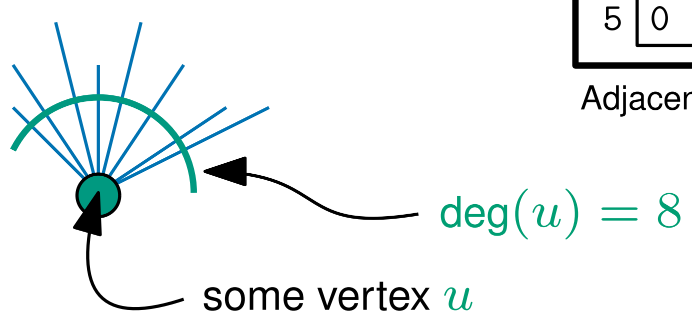
V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time
List all the edges leaving $u \in V$		

$\deg(u)$ *(the degree of u),*
is the number of edges leaving u



$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 1 & 1 & 0 & 1 & 1 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 0 & 1 & 1 & 1 & 0
\end{array}
$$

Adjacency Matrix

$1 \to 2 \to 3$
$2 \to 1 \to 3 \to 4$
$3 \to 1 \to 2$
$4 \to 2 \to 3$
$5 \to 2$

Adjacency List
(using linked lists)

V is the set of vertices
$|V|$ *is the number of vertices*

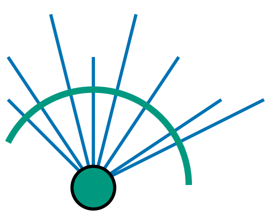
Basic operations

E is the set of edges
$|E|$ *is the number of edges*

	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time
List all the edges leaving $u \in V$		

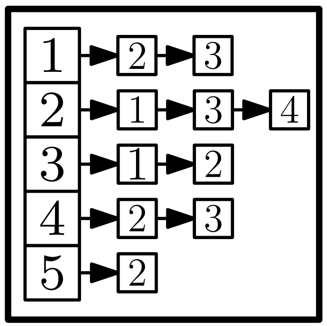
$\deg(u)$ *(the degree of u),*

is the number of edges leaving u



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

Adjacency Matrix



Adjacency List
(using linked lists)

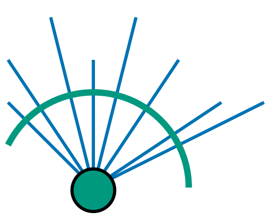
V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

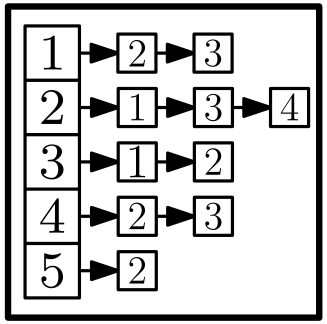
	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time
List all the edges leaving $u \in V$	$O(V)$ time	

$\deg(u)$ *(the degree of u),*
is the number of edges leaving u



	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

Adjacency Matrix



Adjacency List
(using linked lists)

University of BRISTOL

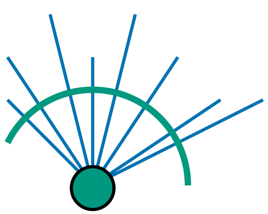
V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

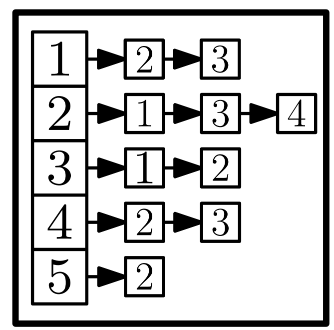
	Adjacency Matrix	Adjacency List *(using linked lists)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time

$\deg(u)$ *(the degree of u),*
is the number of edges leaving u



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 & 1 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 1 & 1 & 1 & 0 \end{array}$$

Adjacency Matrix



Adjacency List
(using linked lists)

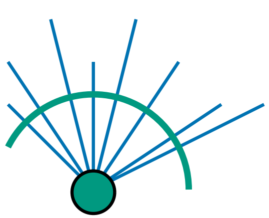
V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

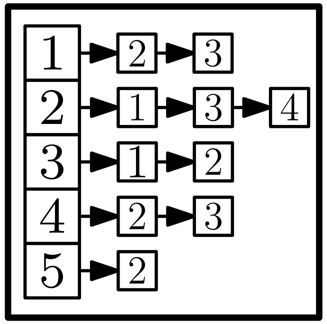
	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

$\deg(u)$ *(the degree of u),*
is the number of edges leaving u

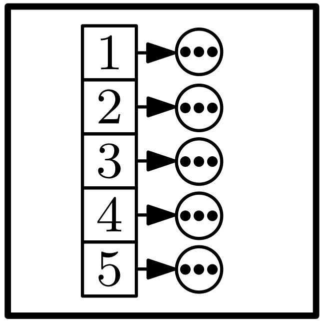


	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

Adjacency Matrix



Adjacency List
(using linked lists)



Adjacency List
(using hash tables)

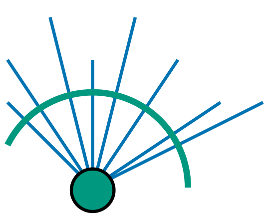
V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

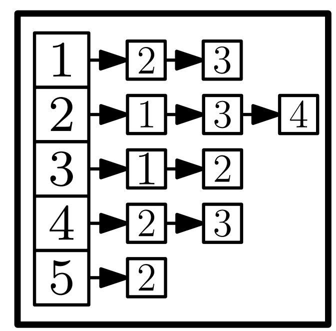
	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\vert V\vert^2)$	$\Theta(\vert V\vert + \vert E\vert)$	$\Theta(\vert V\vert + \vert E\vert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

$\deg(u)$ *(the degree of u),*
is the number of edges leaving u

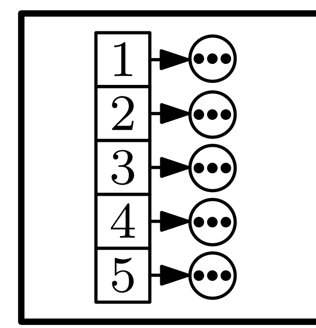


$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 & 1 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 1 & 1 & 1 & 0 \\ \hline \end{array}$$

Adjacency Matrix



Adjacency List
(using linked lists)



Adjacency List
(using hash tables)

all three representations work for directed and/or weighted graphs too

(with the same complexities)

Basic operations

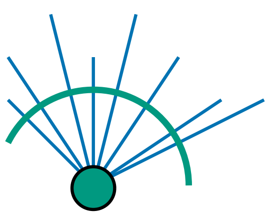
V is the set of vertices
$|V|$ *is the number of vertices*

E is the set of edges
$|E|$ *is the number of edges*

why don't we always use these?

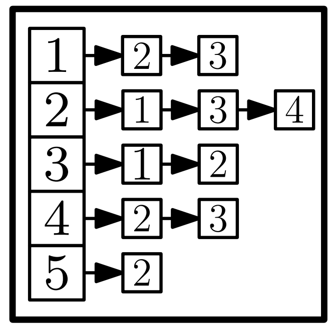
	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

$\deg(u)$ *(the degree of u),*
is the number of edges leaving u

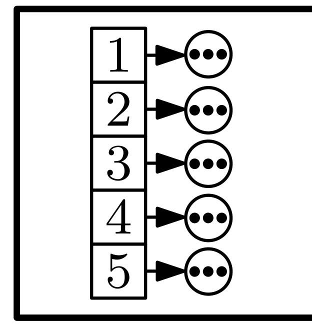


$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 1 & 1 & 0 & 1 & 1 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 0 & 1 & 1 & 1 & 0 \\
\hline
\end{array}
$$

Adjacency Matrix



Adjacency List
(using linked lists)



Adjacency List
(using hash tables)

all three representations work for directed and/or weighted graphs too
(with the same complexities)

V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

why don't we always use these?

	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

why don't we always use these?

	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

Basic hash tables give *expected* time complexities (constant time 'on average')

- no worst case guarantees against collisions

V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

why don't we always use these?

	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

Basic hash tables give *expected* time complexities (constant time 'on average')

- no worst case guarantees against collisions

If you're interested in hashing with provable guarantees,
come to **COMS31900** (Advanced Algorithms)

V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

why don't we always use these?

	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

Basic hash tables give *expected* time complexities (constant time 'on average')

- no worst case guarantees against collisions

If you're interested in hashing with provable guarantees,
come to **COMS31900** (Advanced Algorithms)

Fortunately, few algorithms need to ask *"is there an edge?"*

Normally, *"tell me all the edges"* is fine

V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

why don't we always use these?

	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

why don't we always use these?

	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

For this course we'll stick to Adjacency Matrices and Lists (using linked lists)

(for today just Adjacency Lists)

V is the set of vertices

$|V|$ *is the number of vertices*

Basic operations

E is the set of edges

$|E|$ *is the number of edges*

why don't we always use these?

	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

For this course we'll stick to Adjacency Matrices and Lists (using linked lists)

(for today just Adjacency Lists)

One practical reason to stick to standard representations is that

you may not have control over how your graph is stored.

V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

why don't we always use these?

	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\lvert V\rvert^2)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert + \lvert E\rvert)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

For this course we'll stick to Adjacency Matrices and Lists (using linked lists)

(for today just Adjacency Lists)

One practical reason to stick to standard representations is that

you may not have control over how your graph is stored.

it may not even be stored as a graph...

V is the set of vertices
$|V|$ *is the number of vertices*

Basic operations

E is the set of edges
$|E|$ *is the number of edges*

why don't we always use these?

	Adjacency Matrix	Adjacency List *(using linked lists)*	Adjacency List *(using hash tables)*
Space	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$
Is there an edge from vertex u to v?	$O(1)$ time	$O(\deg(u))$ time	$O(1)$ time
List all the edges leaving $u \in V$	$O(V)$ time	$O(\deg(u))$ time	$O(\deg(u))$ time

For this course we'll stick to Adjacency Matrices and Lists (using linked lists)

(for today just Adjacency Lists)

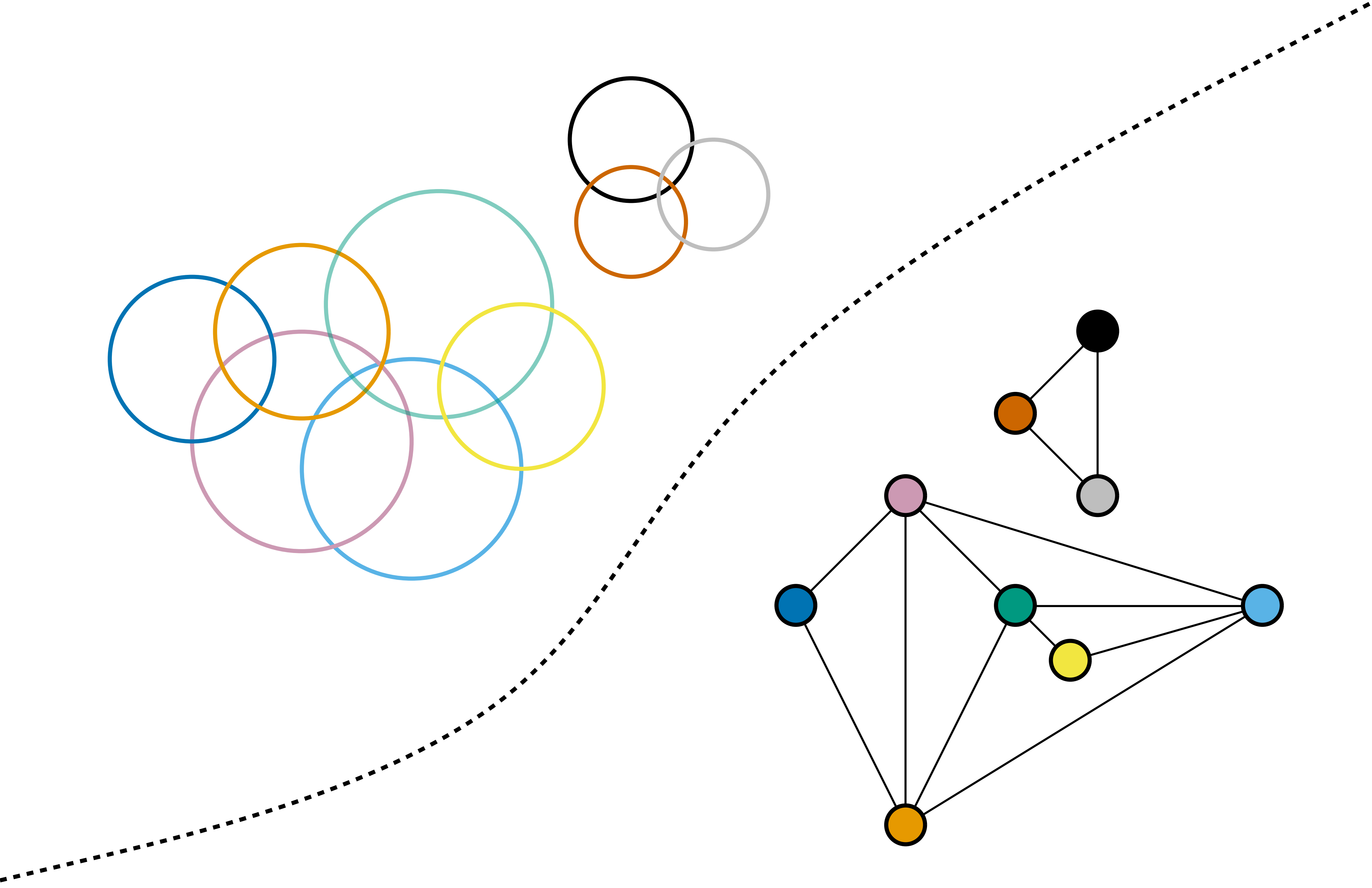
One practical reason to stick to standard representations is that

you may not have control over how your graph is stored.

it may not even be stored as a graph...

Fortunately in many cases, you can *pretend* your graph is stored as one of the above.

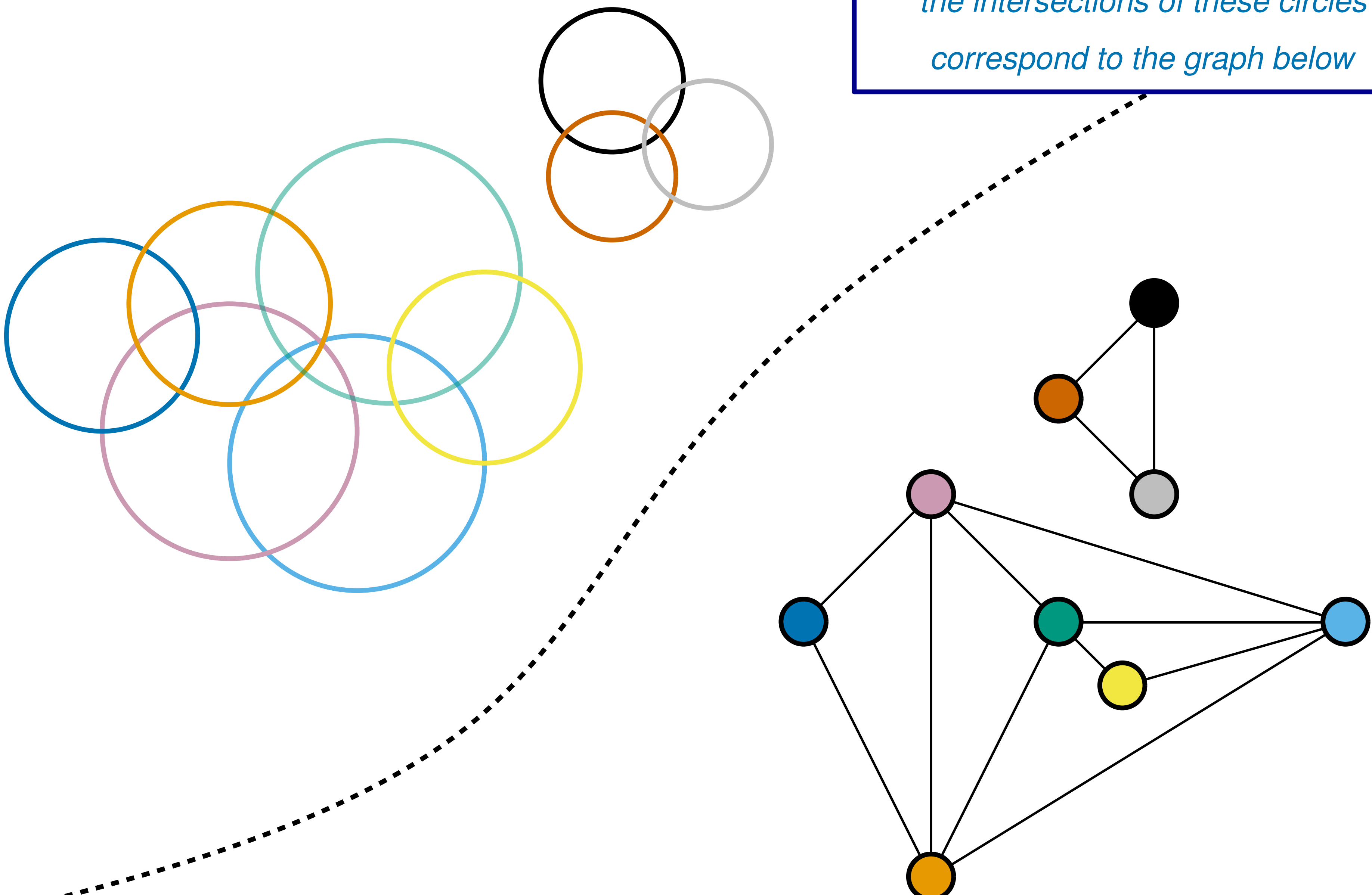
Intersection Graphs



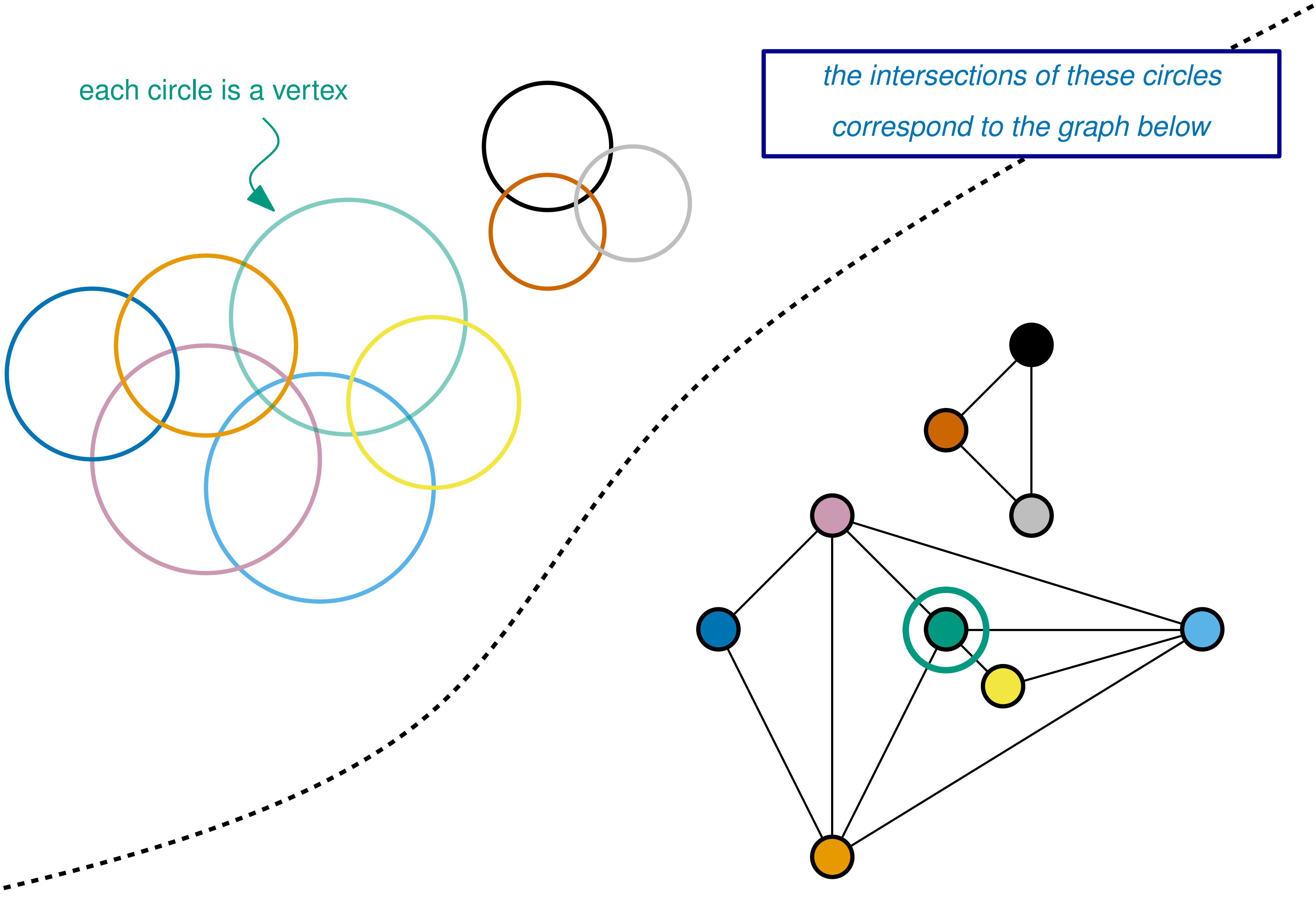
Intersection Graphs

the intersections of these circles

correspond to the graph below



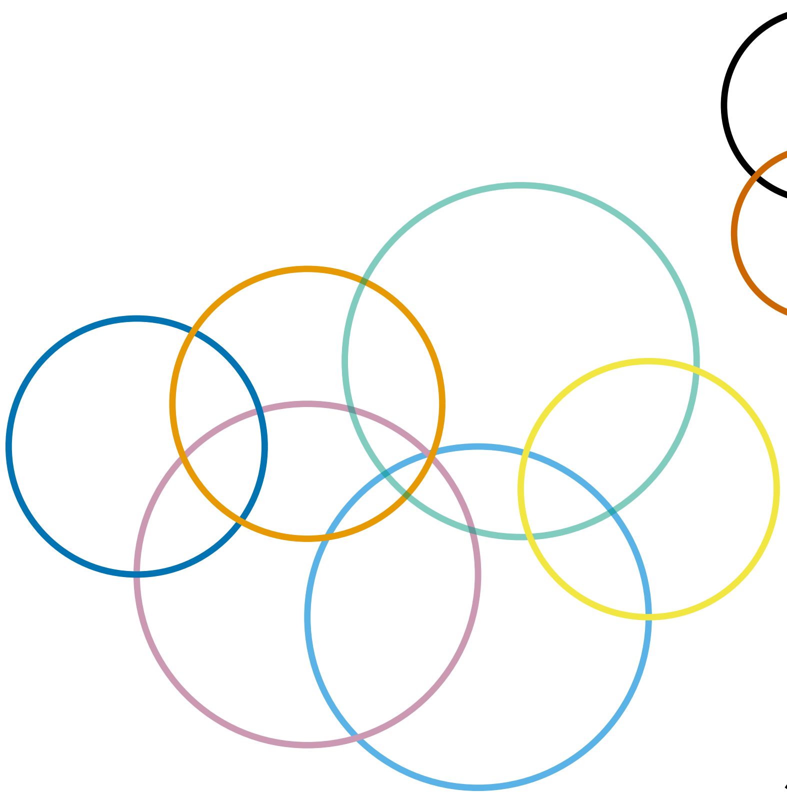
Intersection Graphs

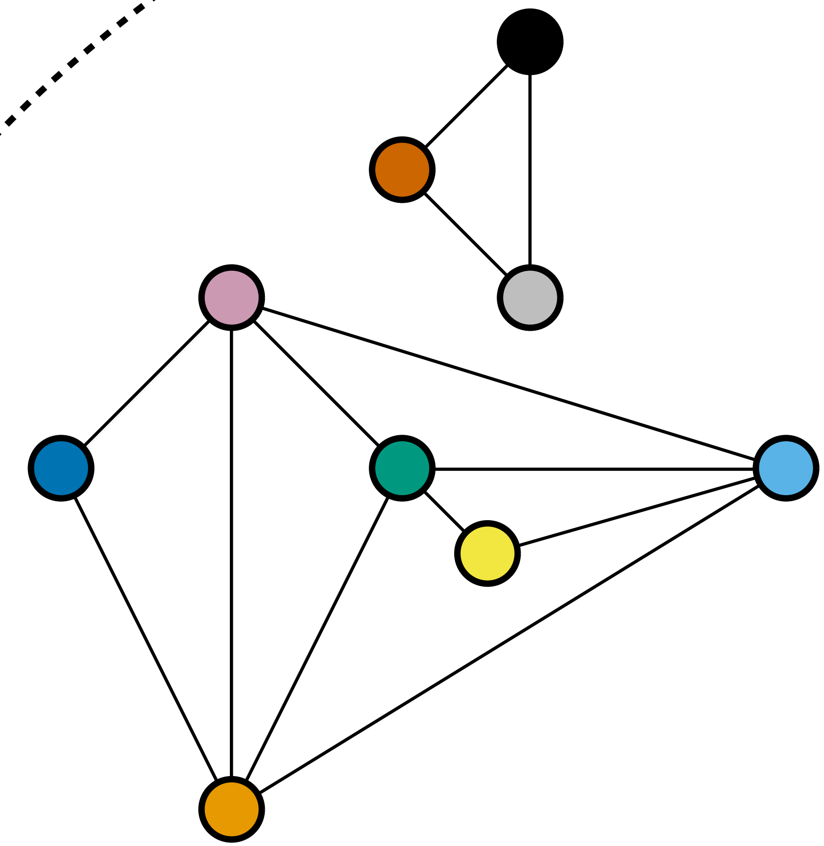


Intersection Graphs

the intersections of these circles

correspond to the graph below





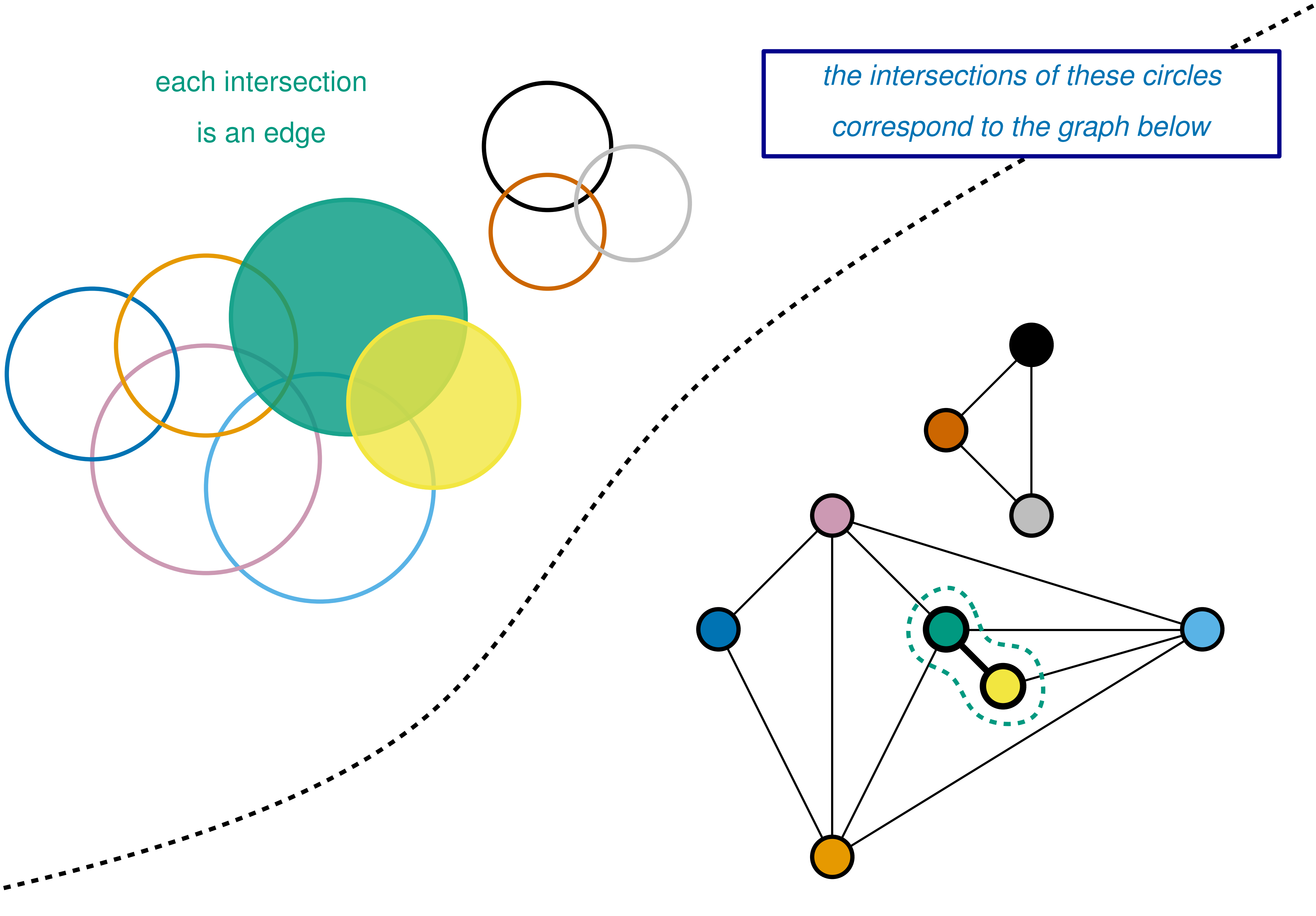
Intersection Graphs

each intersection

is an edge

the intersections of these circles

correspond to the graph below



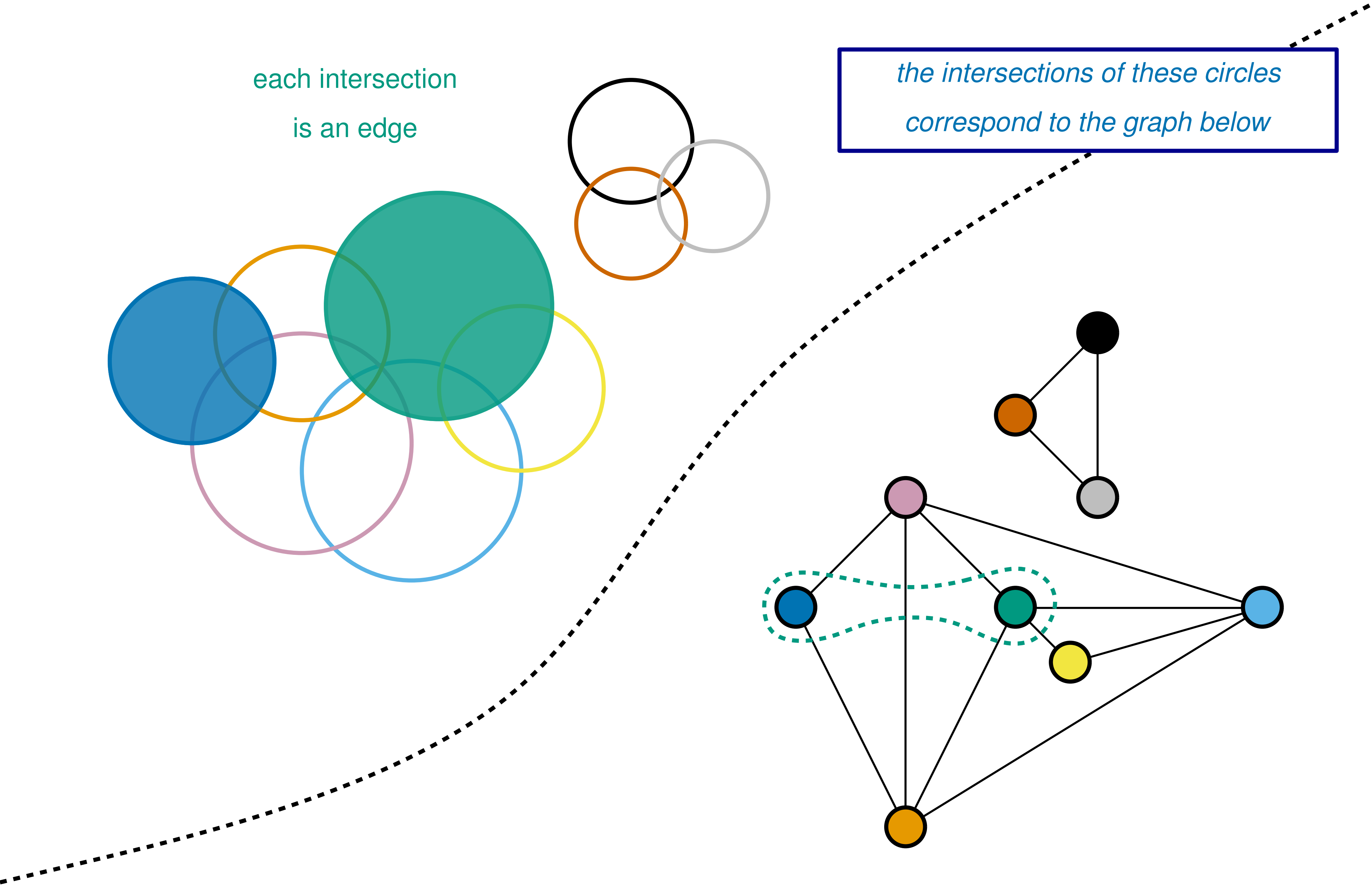
Intersection Graphs

each intersection

is an edge

the intersections of these circles

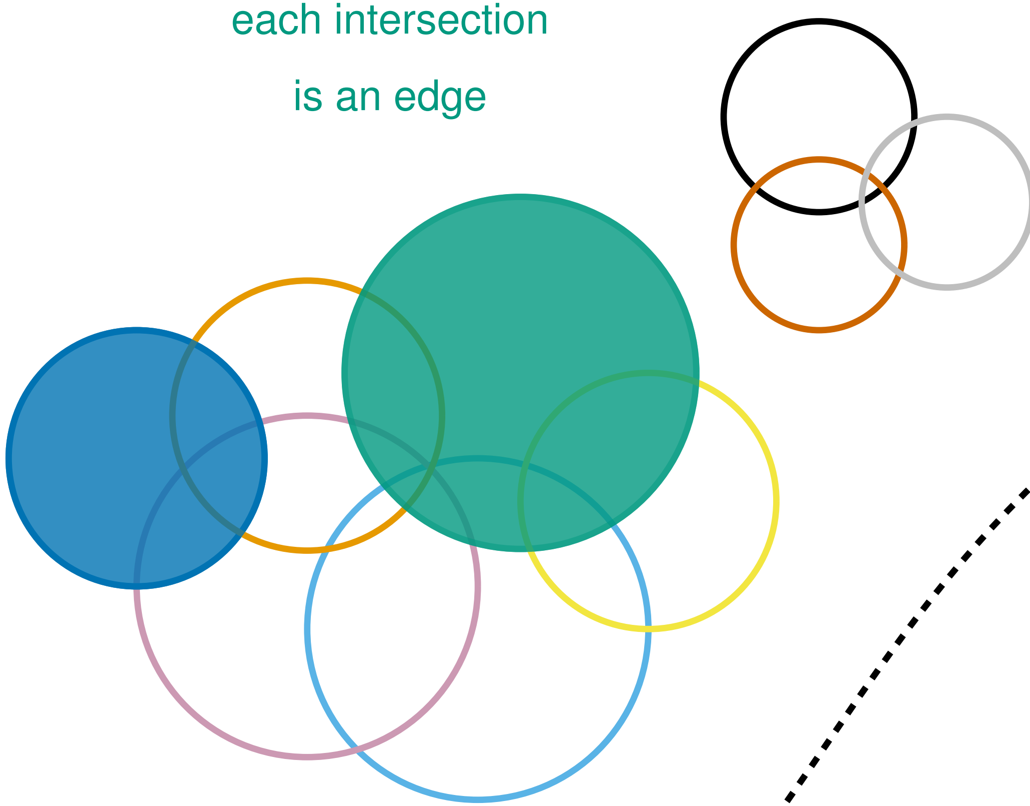
correspond to the graph below



Intersection Graphs

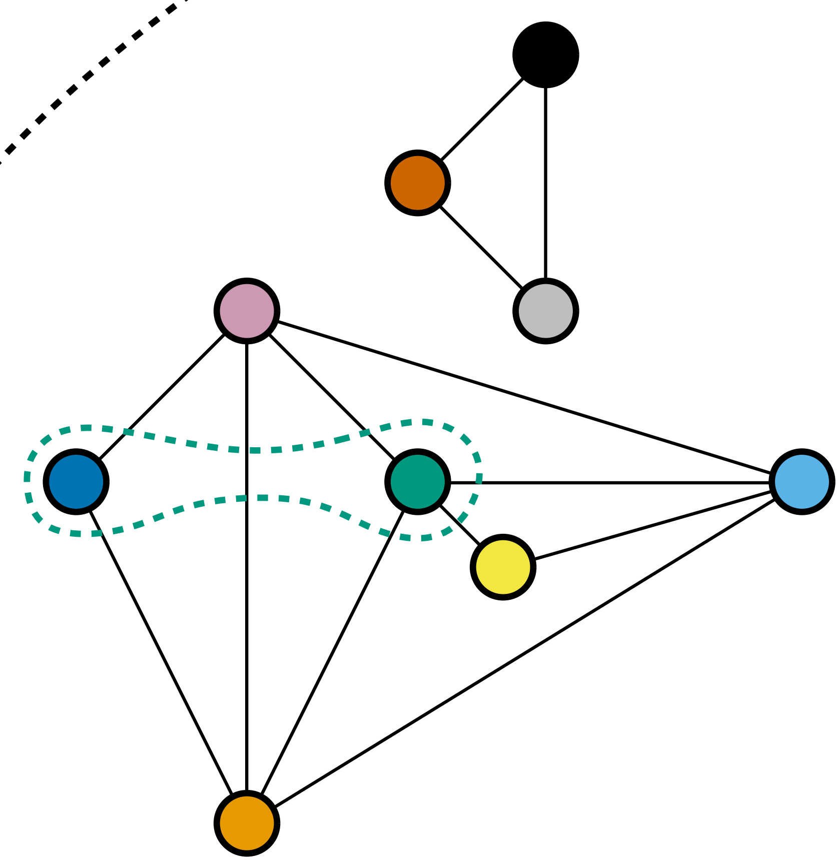
each intersection

is an edge



the intersections of these circles

correspond to the graph below



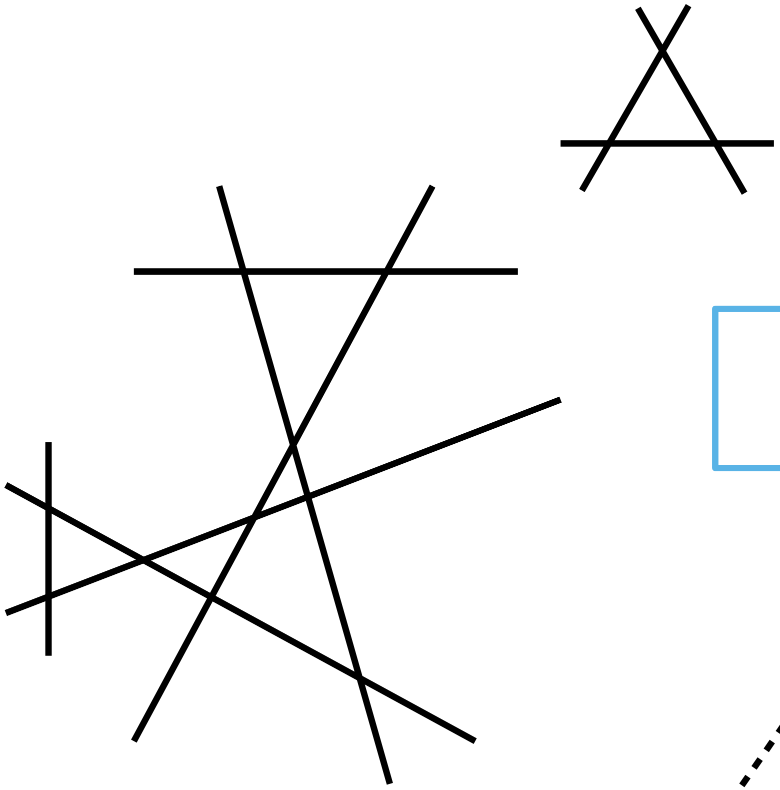
even if we only store the circles,

we can pretend we have

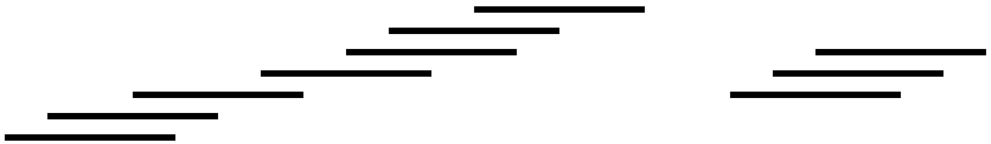
an **Adjacency Matrix**

without building one

Other intersection graphs...



both of these 'collections' also correspond to the example graph

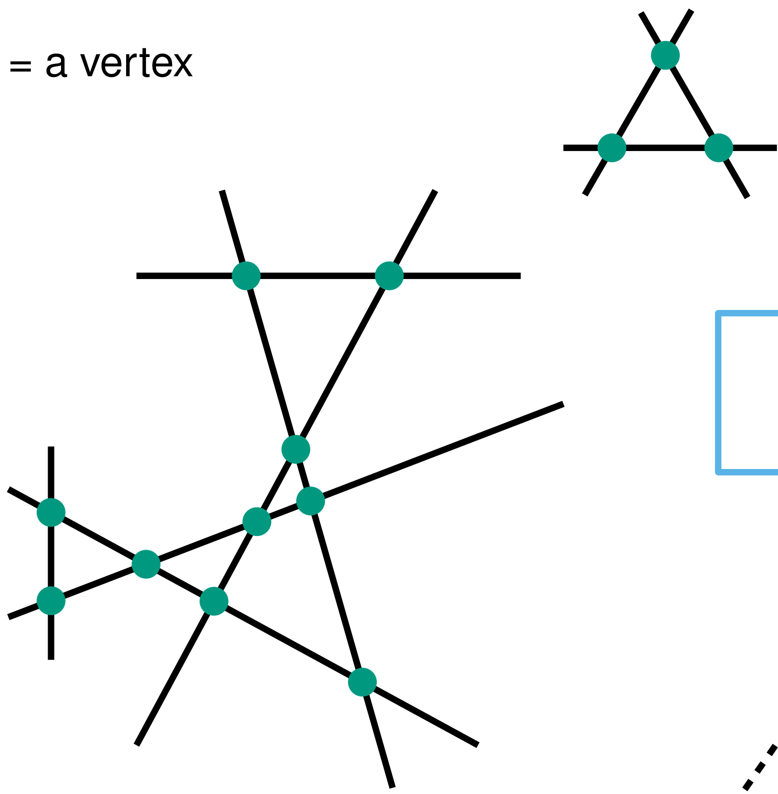


(these are 1D intervals on a line spread out for visual clarity)

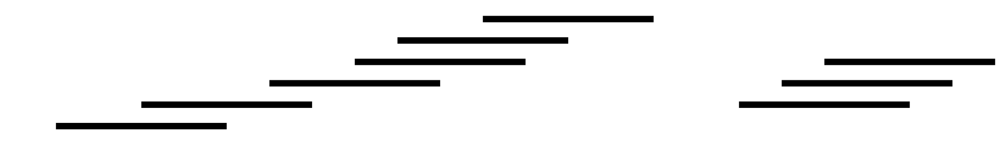
Other intersection graphs. . .

● = an edge

╲ = a vertex



both of these 'collections' also correspond to the example graph

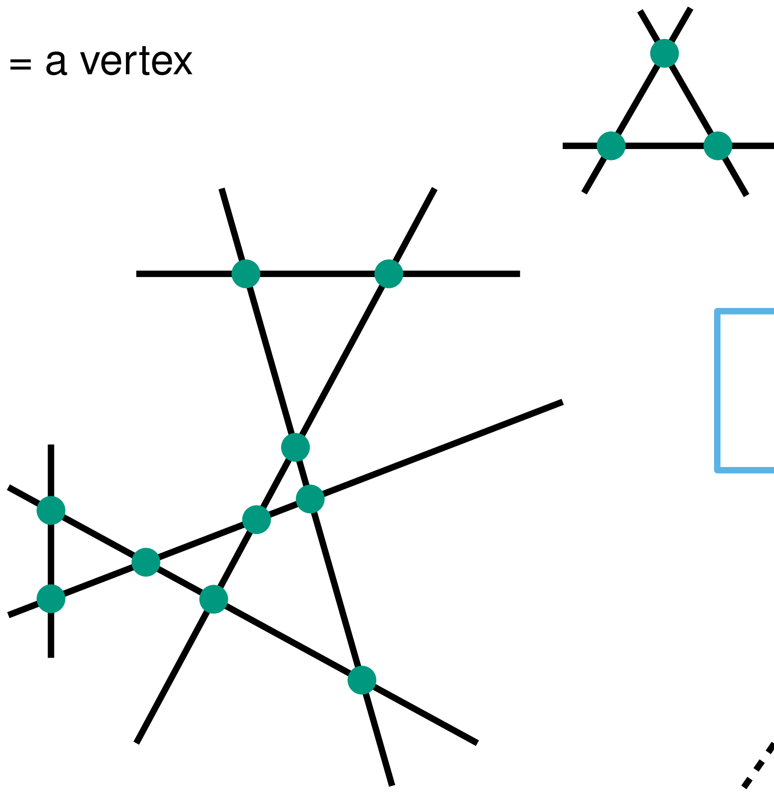


(these are 1D intervals on a line spread out for visual clarity)

Other intersection graphs. . .

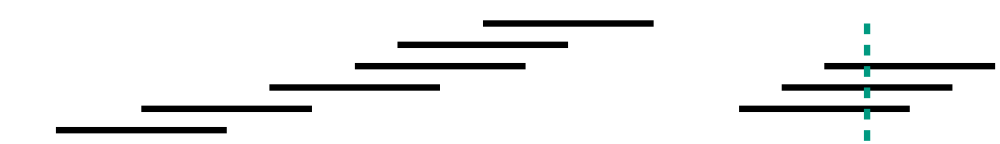
● = an edge

╲ = a vertex



both of these ‘collections’ also correspond to the example graph

there is an edge if two intervals intersect



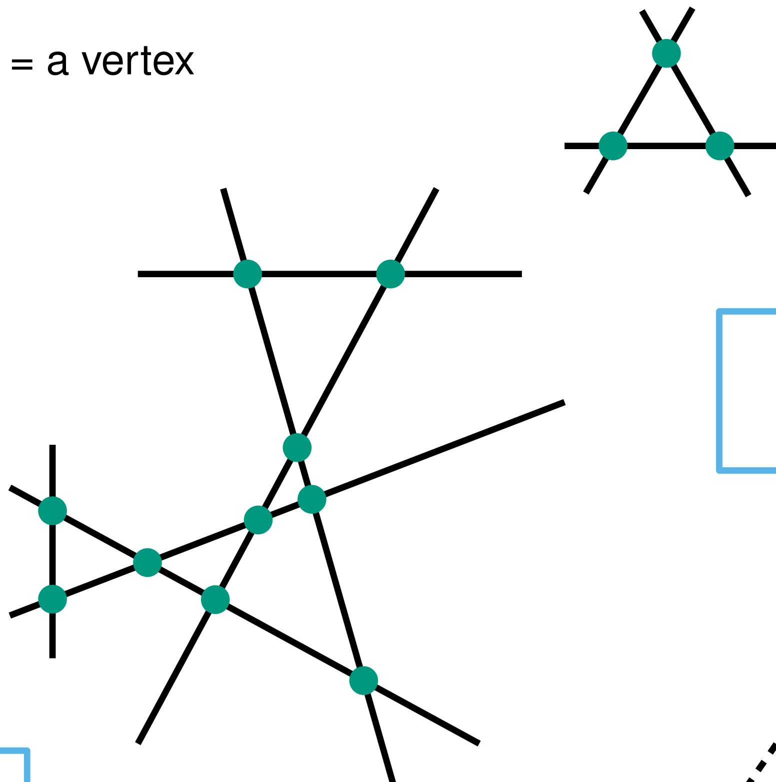
each interval is a vertex

(these are 1D intervals on a line spread out for visual clarity)

Other intersection graphs. . .

● = an edge

\ = a vertex

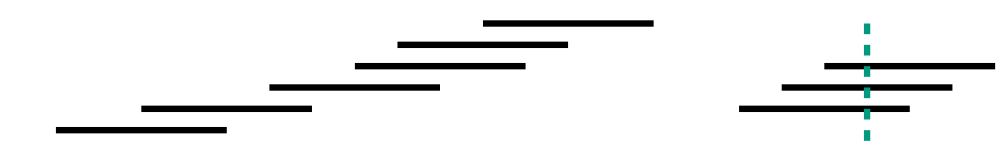


both of these ‘collections’ also correspond to the example graph

Again,

we can pretend we have an **Adjacency Matrix** without building one

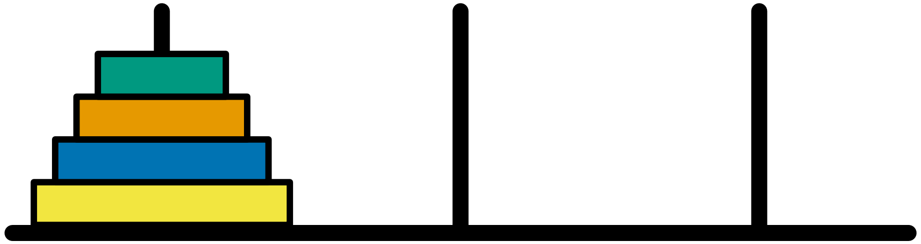
there is an edge if two intervals intersect



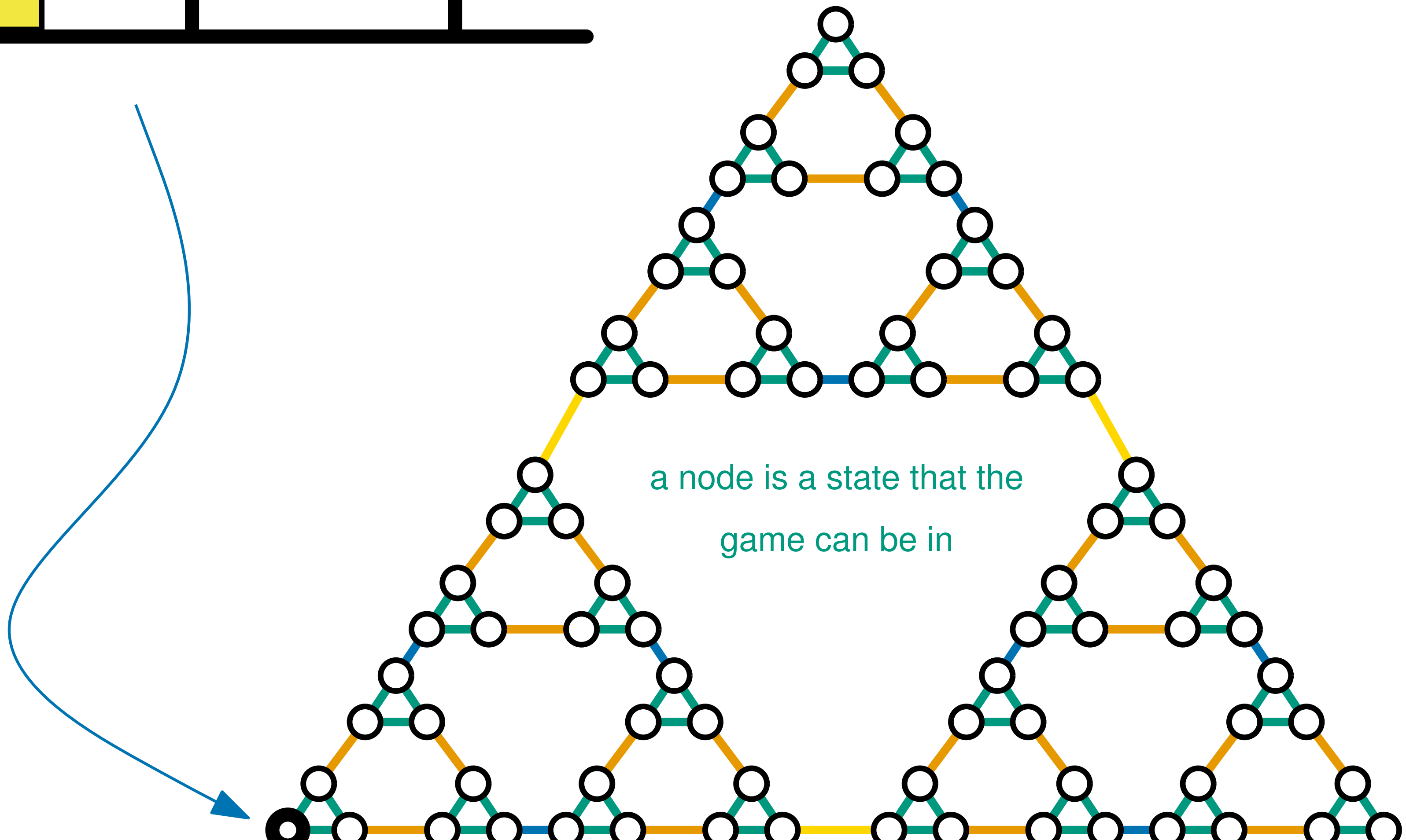
each interval is a vertex

(these are 1D intervals on a line spread out for visual clarity)

Configuration Graphs



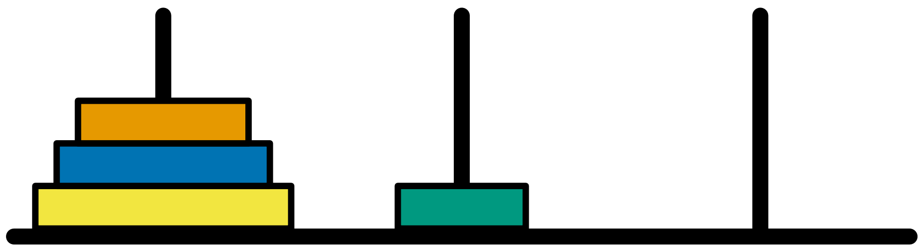
This is the configuration graph for the (4 disk) towers of Hanoi



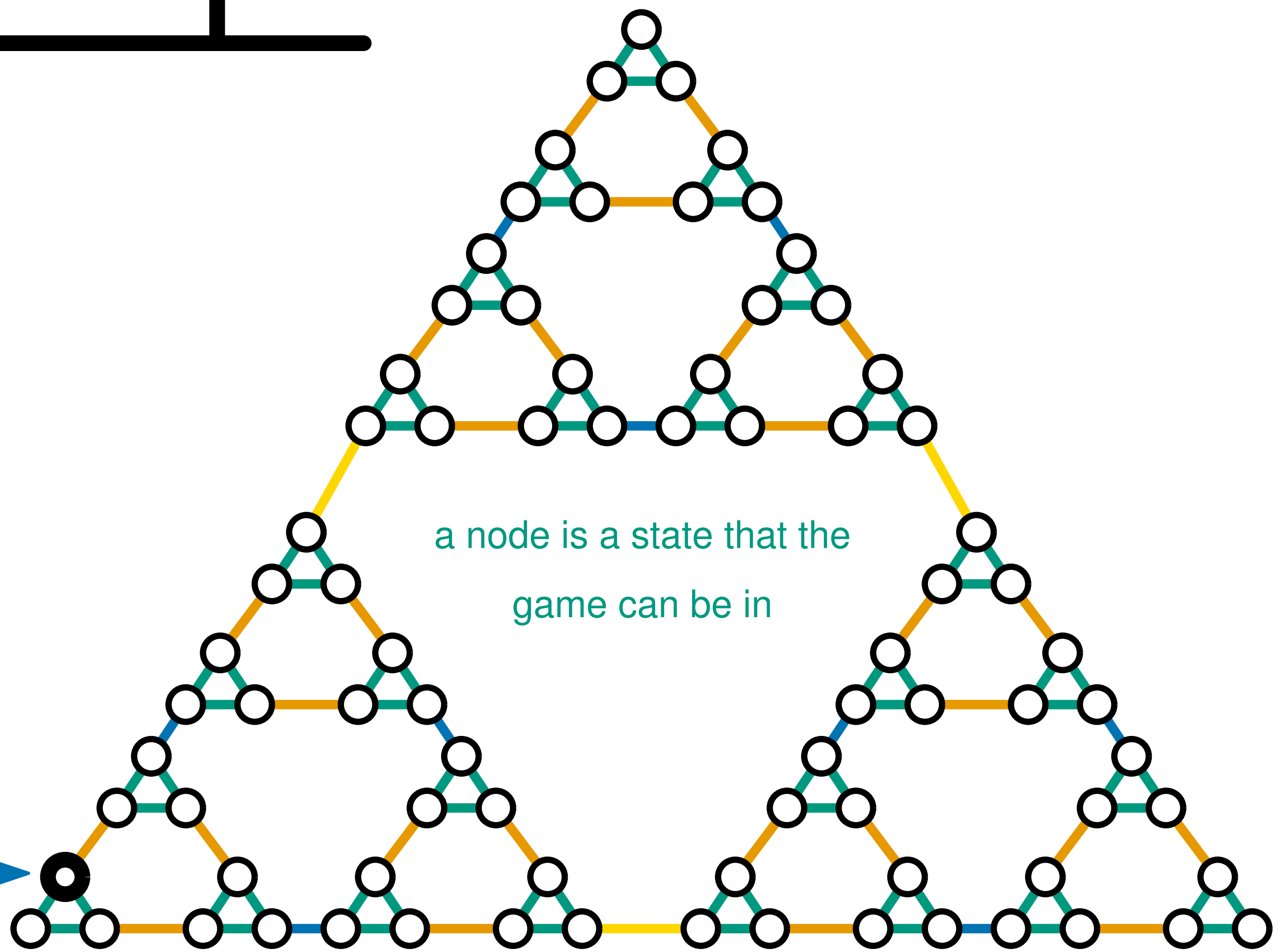
a node is a state that the game can be in

there is an edge if you can get from one state to another in a single move

Configuration Graphs



This is the configuration graph for the (4 disk) towers of Hanoi

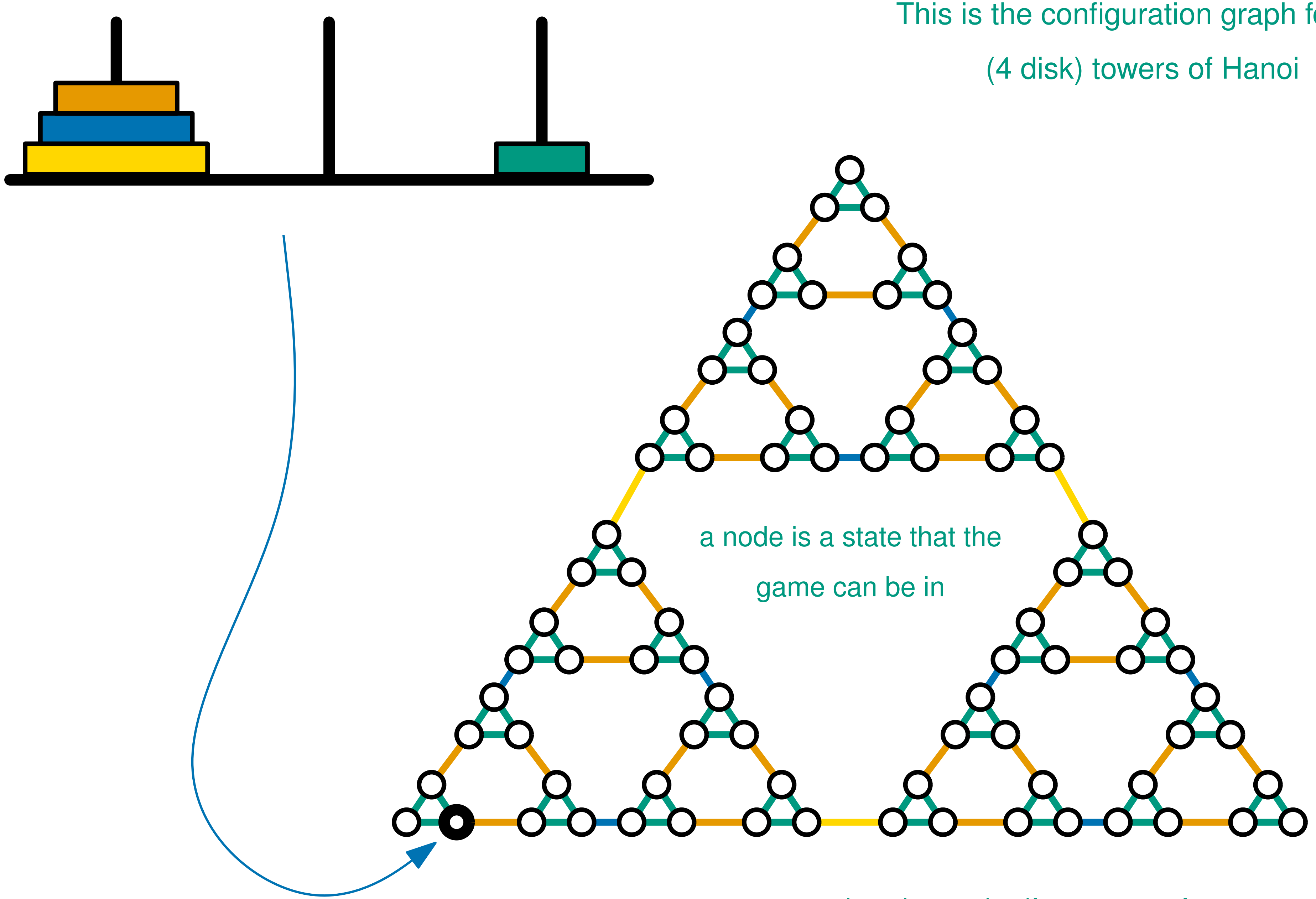


a node is a state that the game can be in

there is an edge if you can get from one state to another in a single move

Configuration Graphs

This is the configuration graph for the (4 disk) towers of Hanoi

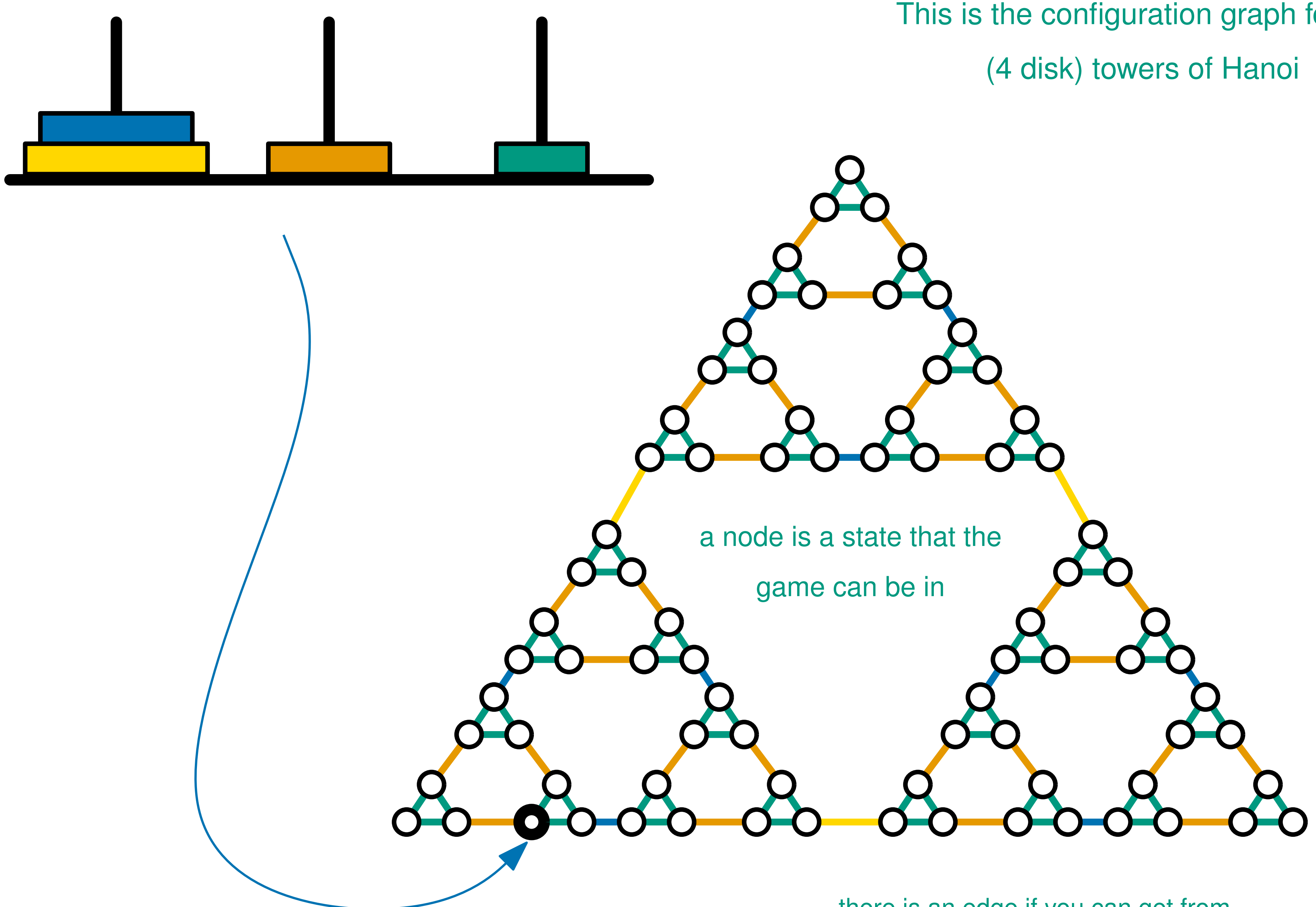


a node is a state that the game can be in

there is an edge if you can get from one state to another in a single move

Configuration Graphs

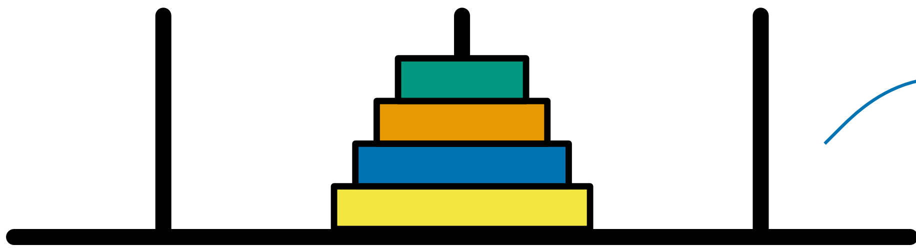
This is the configuration graph for the (4 disk) towers of Hanoi



a node is a state that the game can be in

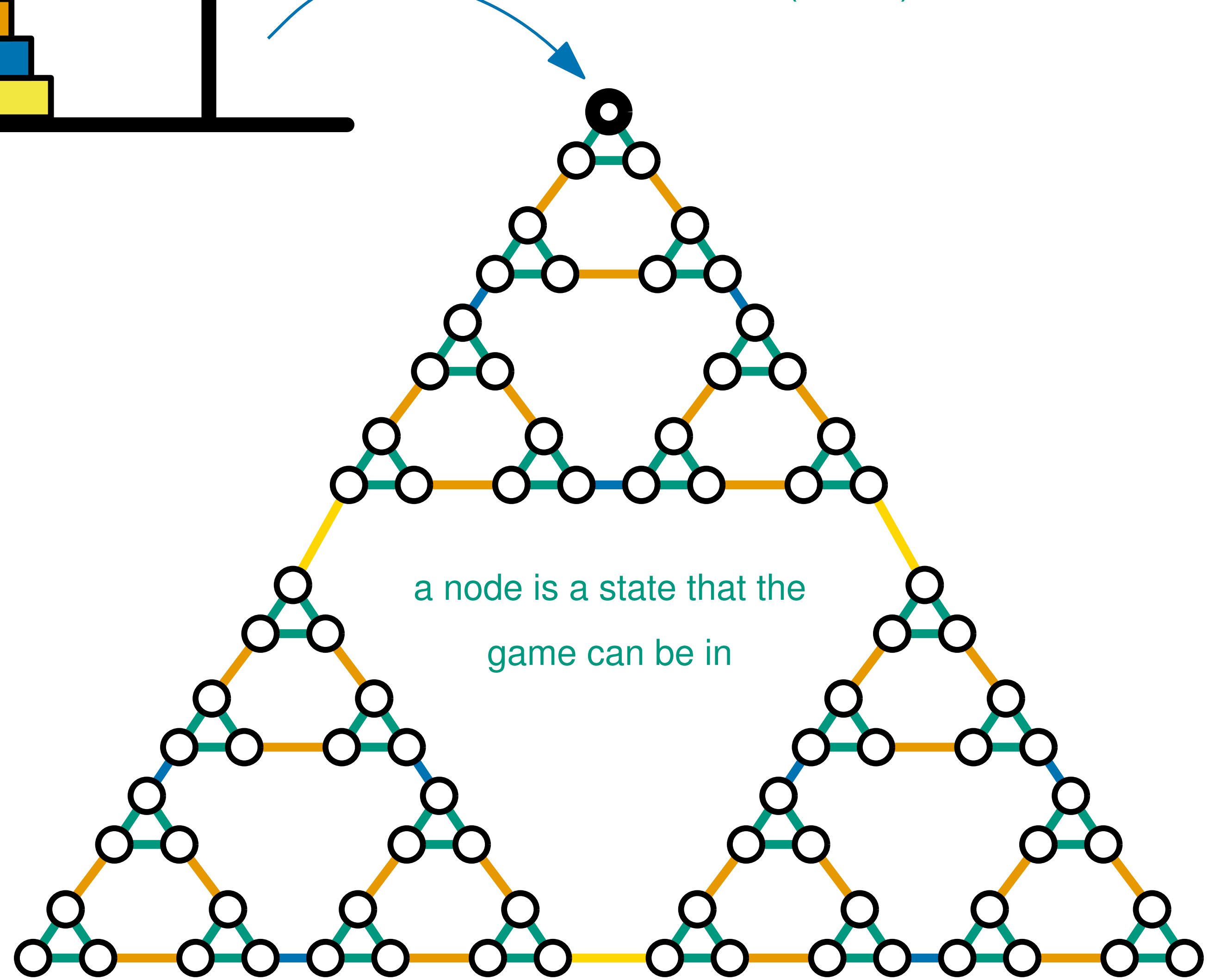
there is an edge if you can get from one state to another in a single move

Configuration Graphs



This is the configuration graph for the

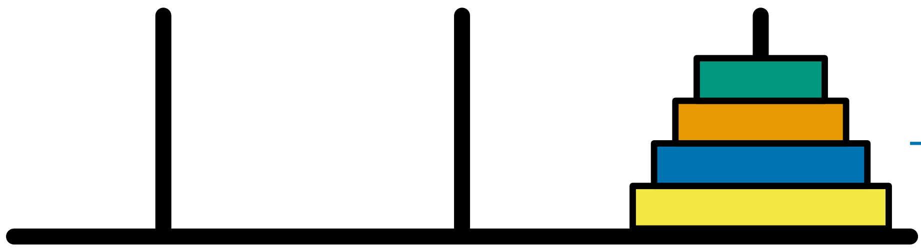
(4 disk) towers of Hanoi



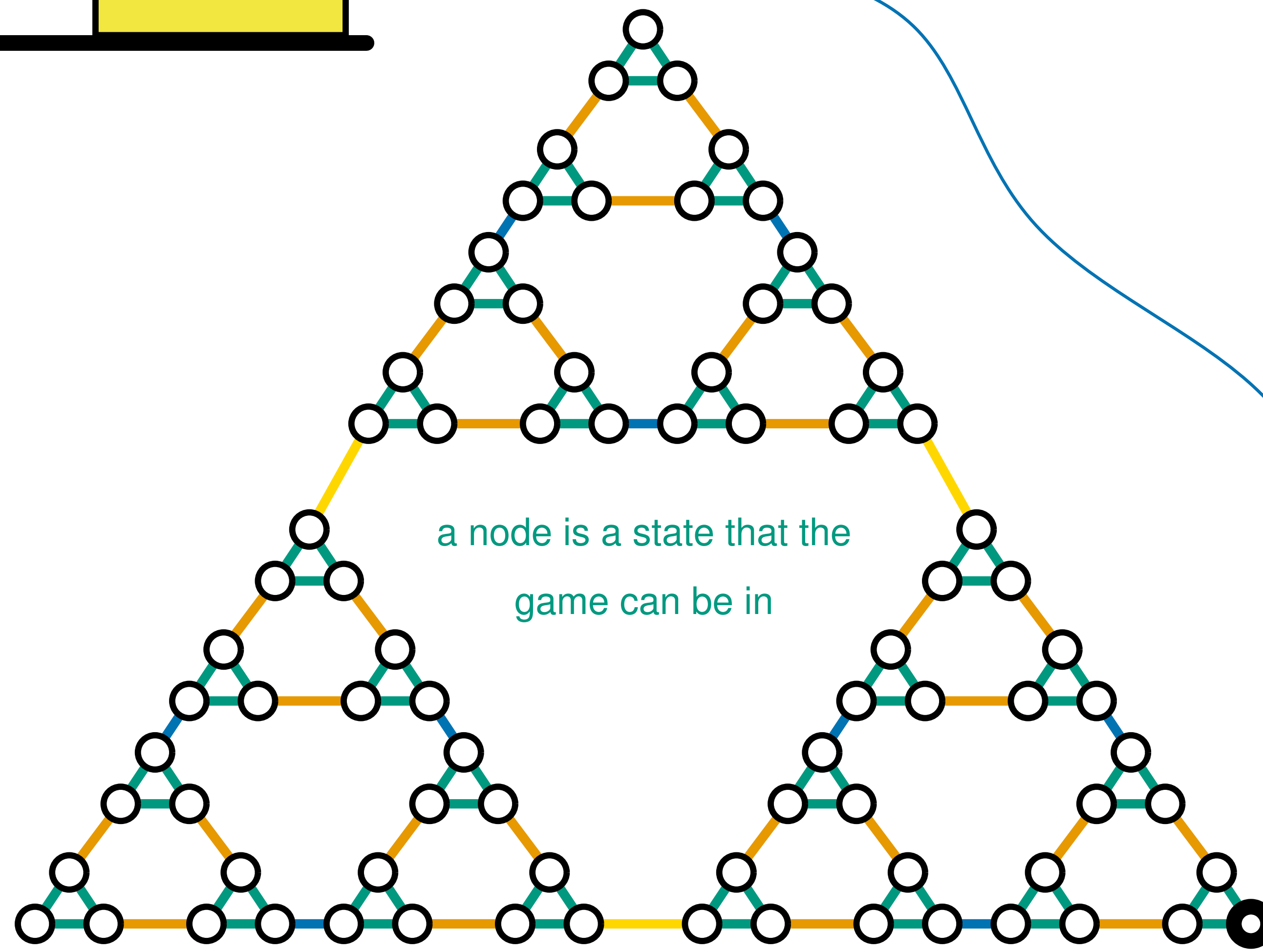
there is an edge if you can get from

one state to another in a single move

Configuration Graphs



This is the configuration graph for the (4 disk) towers of Hanoi



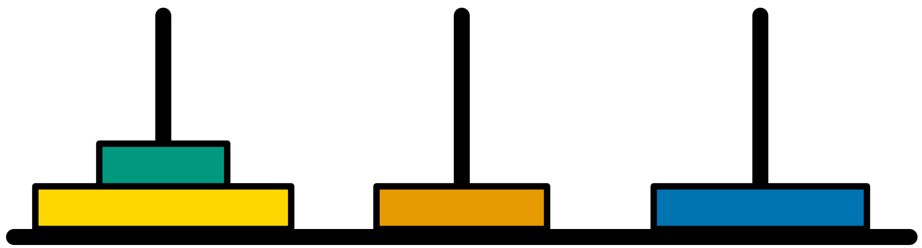
a node is a state that the game can be in

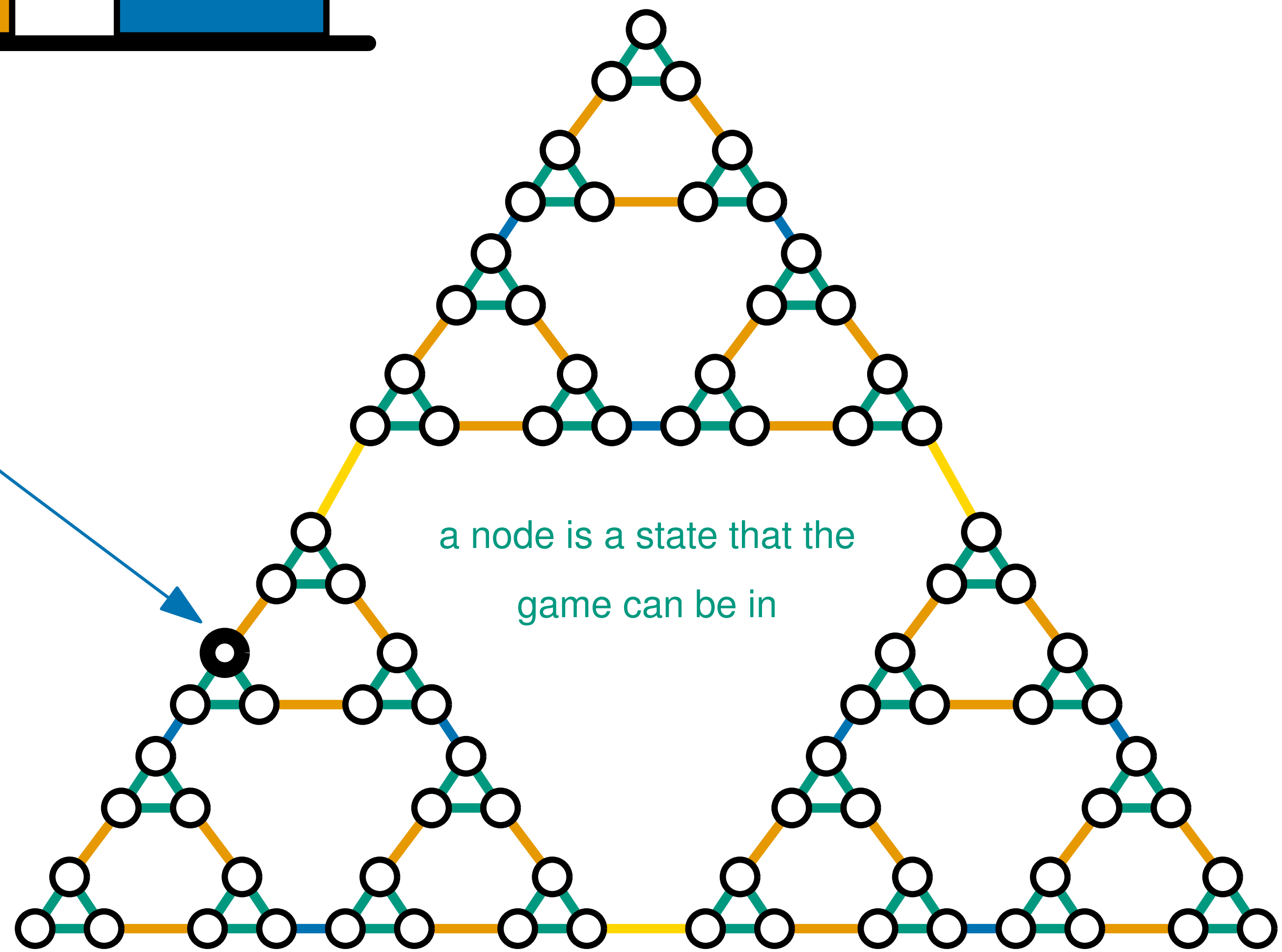
there is an edge if you can get from one state to another in a single move

Configuration Graphs

This is the configuration graph for the

(4 disk) towers of Hanoi

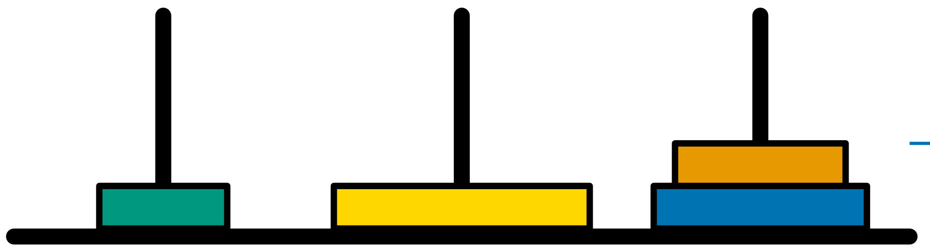




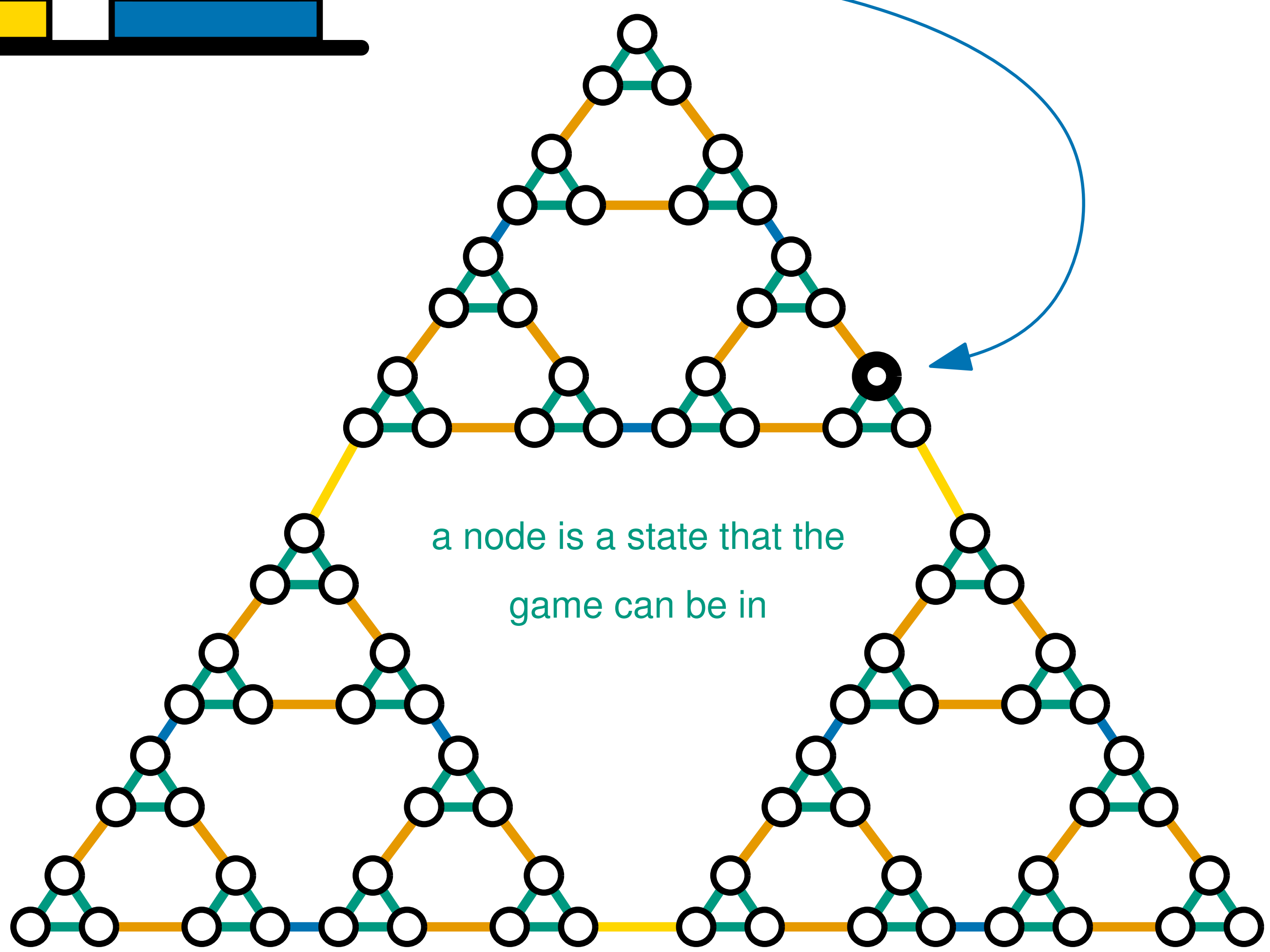
there is an edge if you can get from

one state to another in a single move

Configuration Graphs

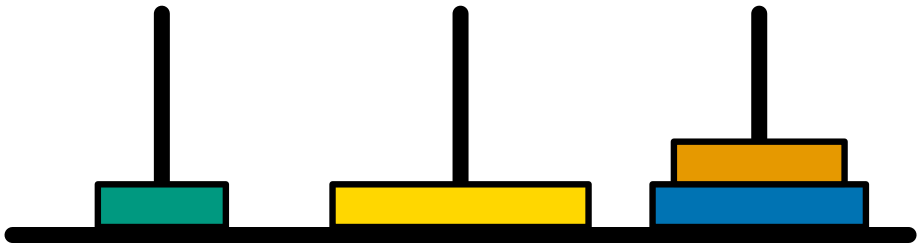


This is the configuration graph for the (4 disk) towers of Hanoi



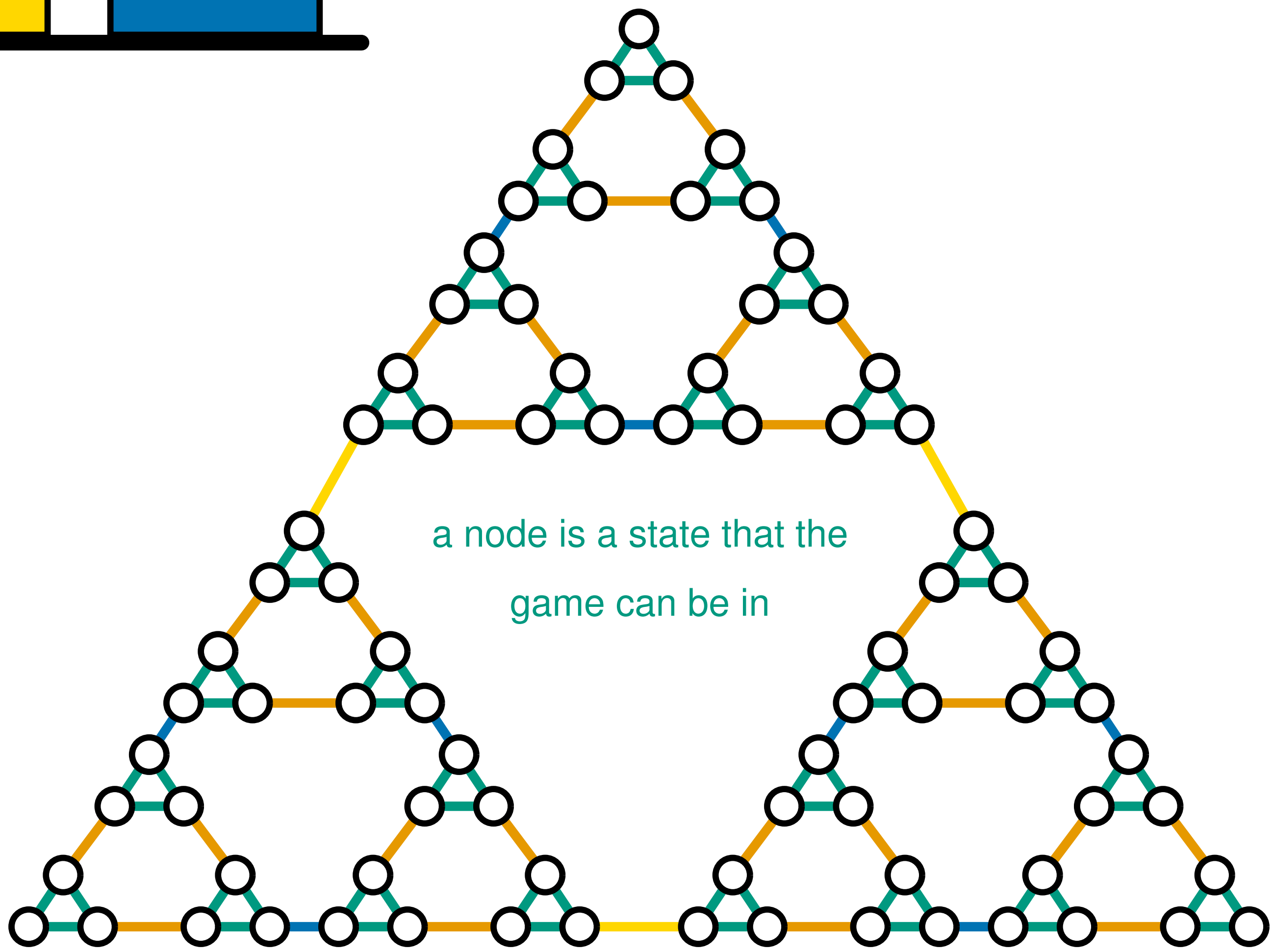
there is an edge if you can get from one state to another in a single move

Configuration Graphs



This is the configuration graph for the

(4 disk) towers of Hanoi



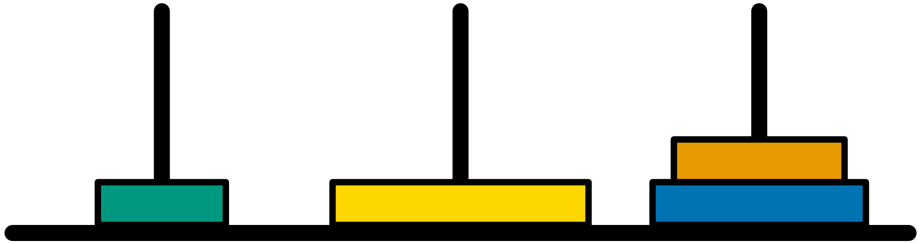
a node is a state that the

game can be in

there is an edge if you can get from

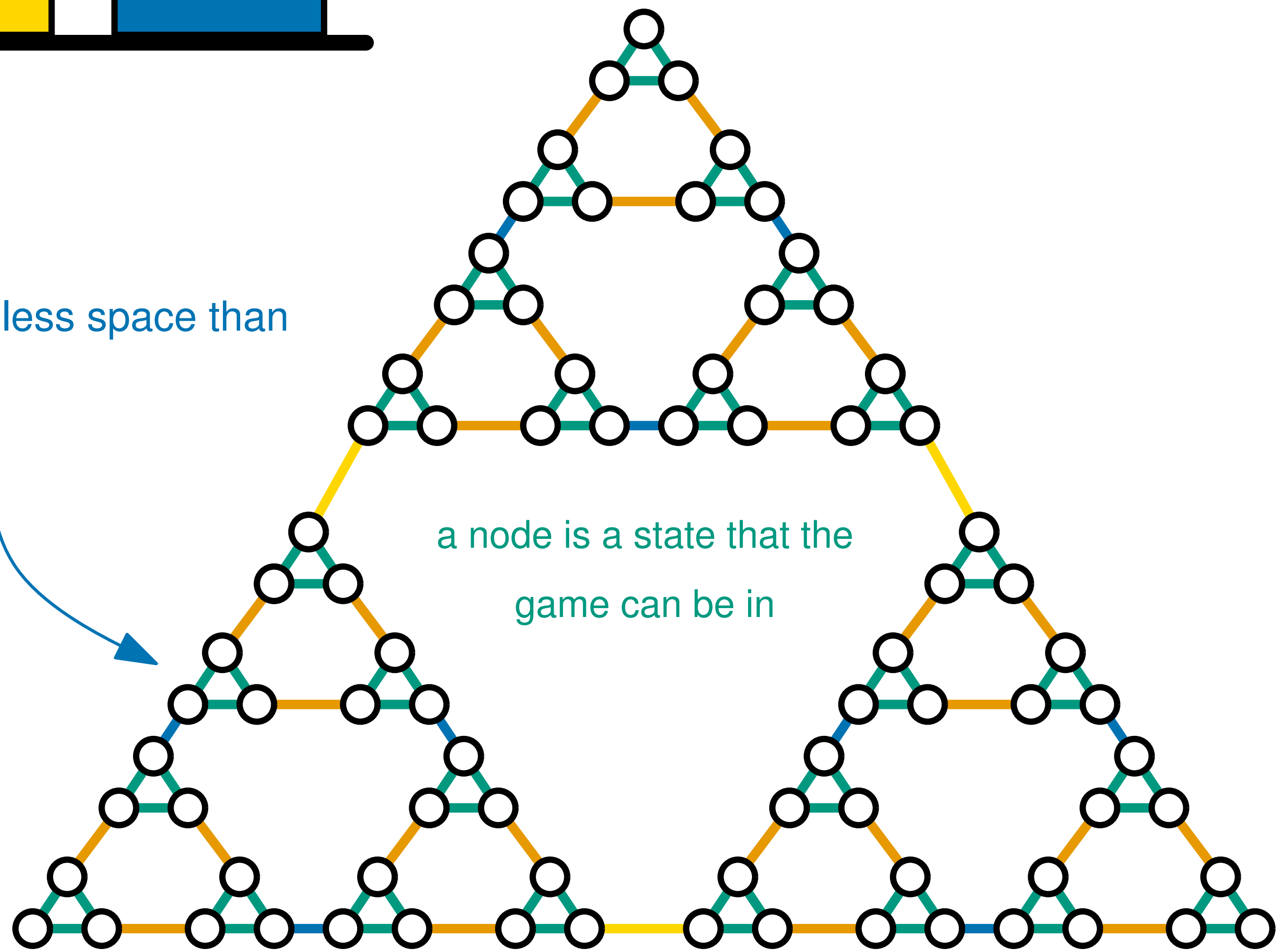
one state to another in a single move

Configuration Graphs



This is the configuration graph for the (4 disk) towers of Hanoi

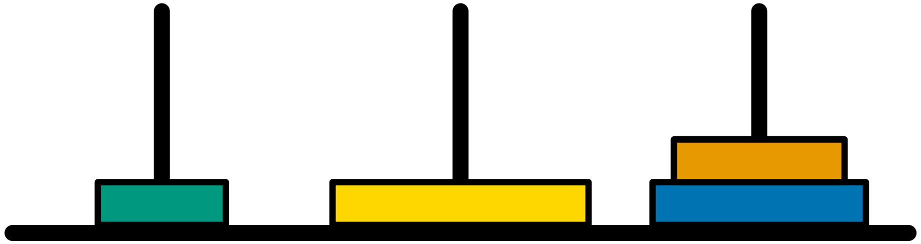
storing **this** uses much less space than storing this **this** graph



a node is a state that the game can be in

there is an edge if you can get from one state to another in a single move

Configuration Graphs

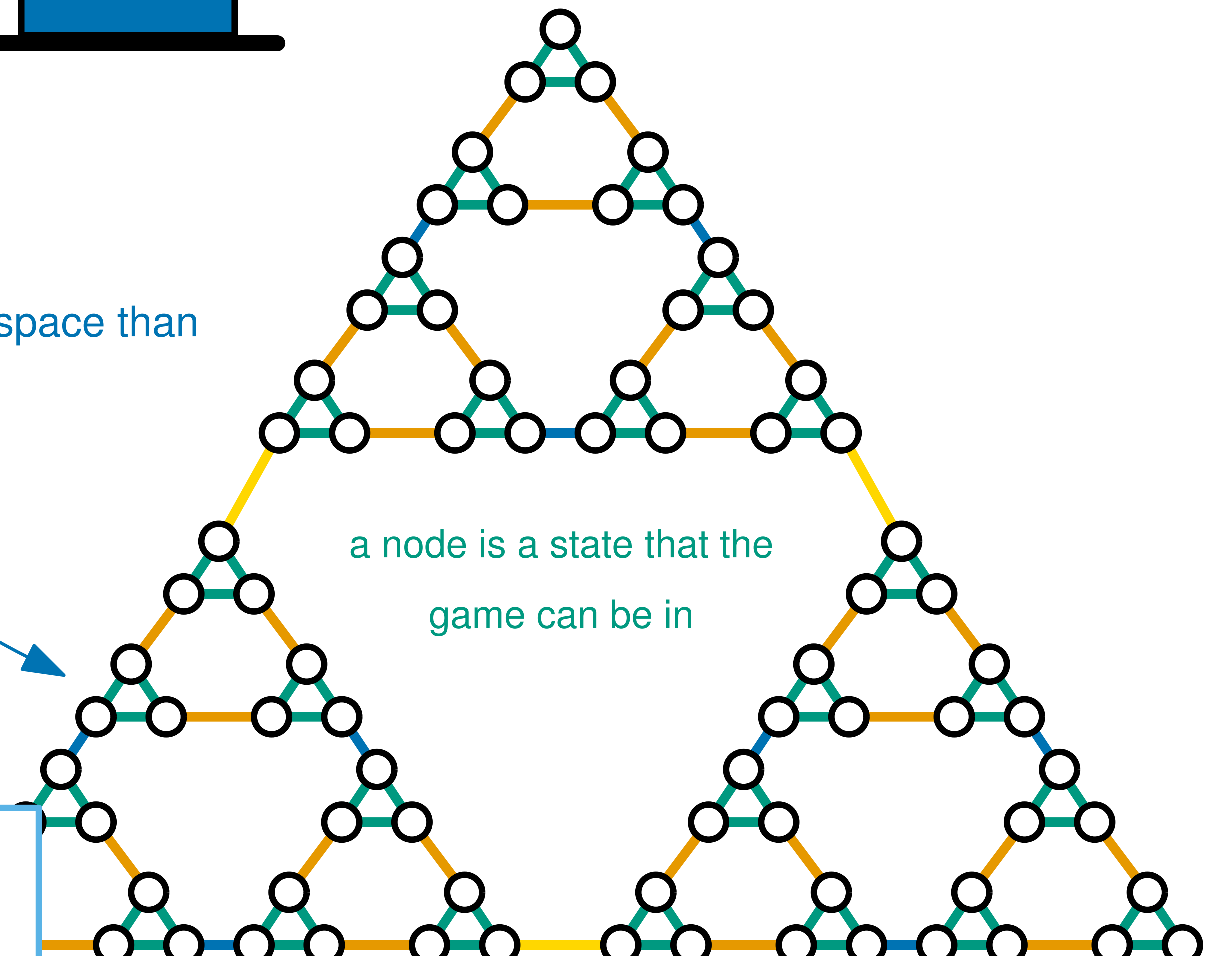


This is the configuration graph for the

(4 disk) towers of Hanoi

storing **this** uses much less space than

storing this **this** graph



a node is a state that the

game can be in

we can pretend we have

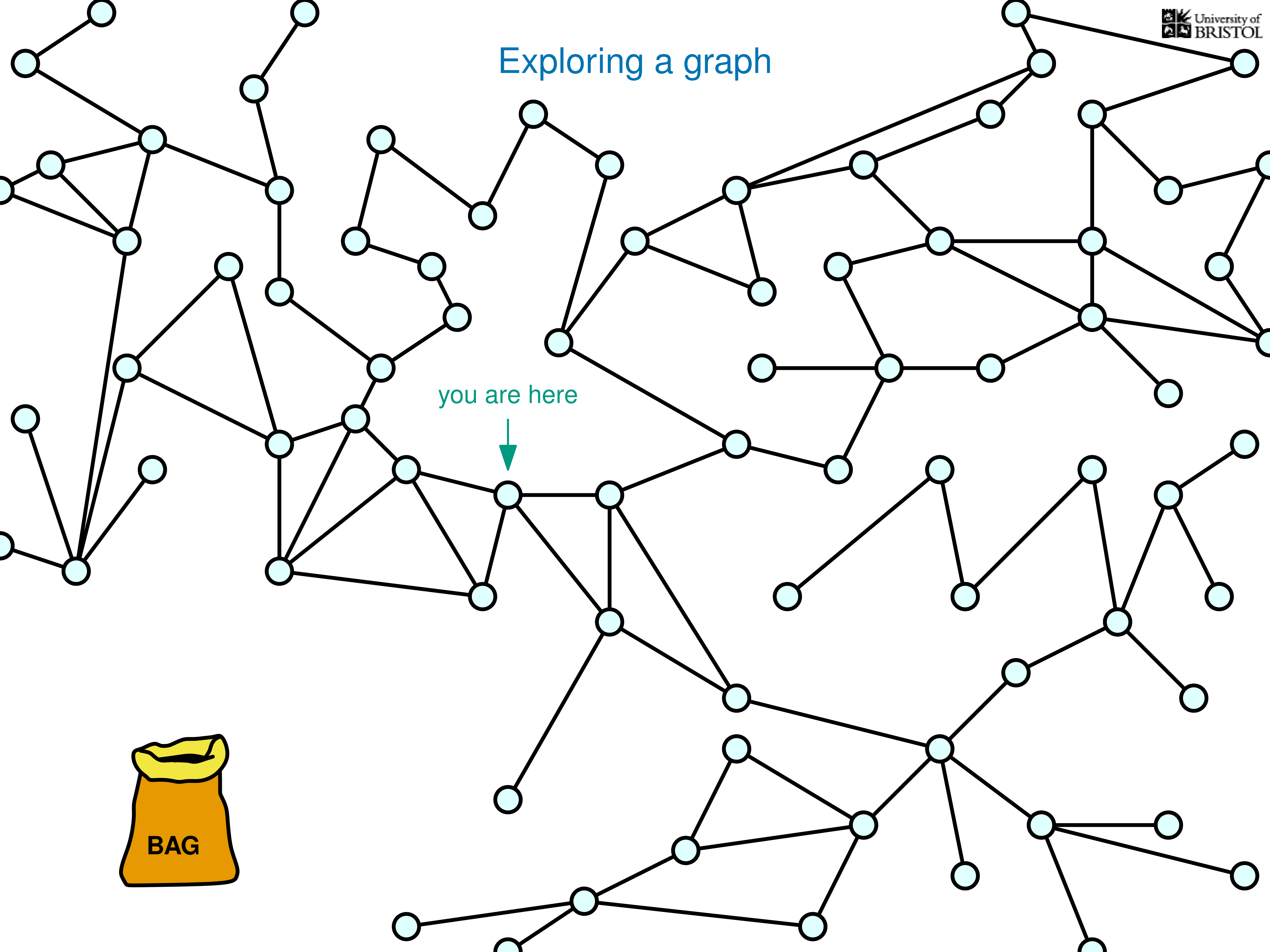
an **Adjacency List**

without building one

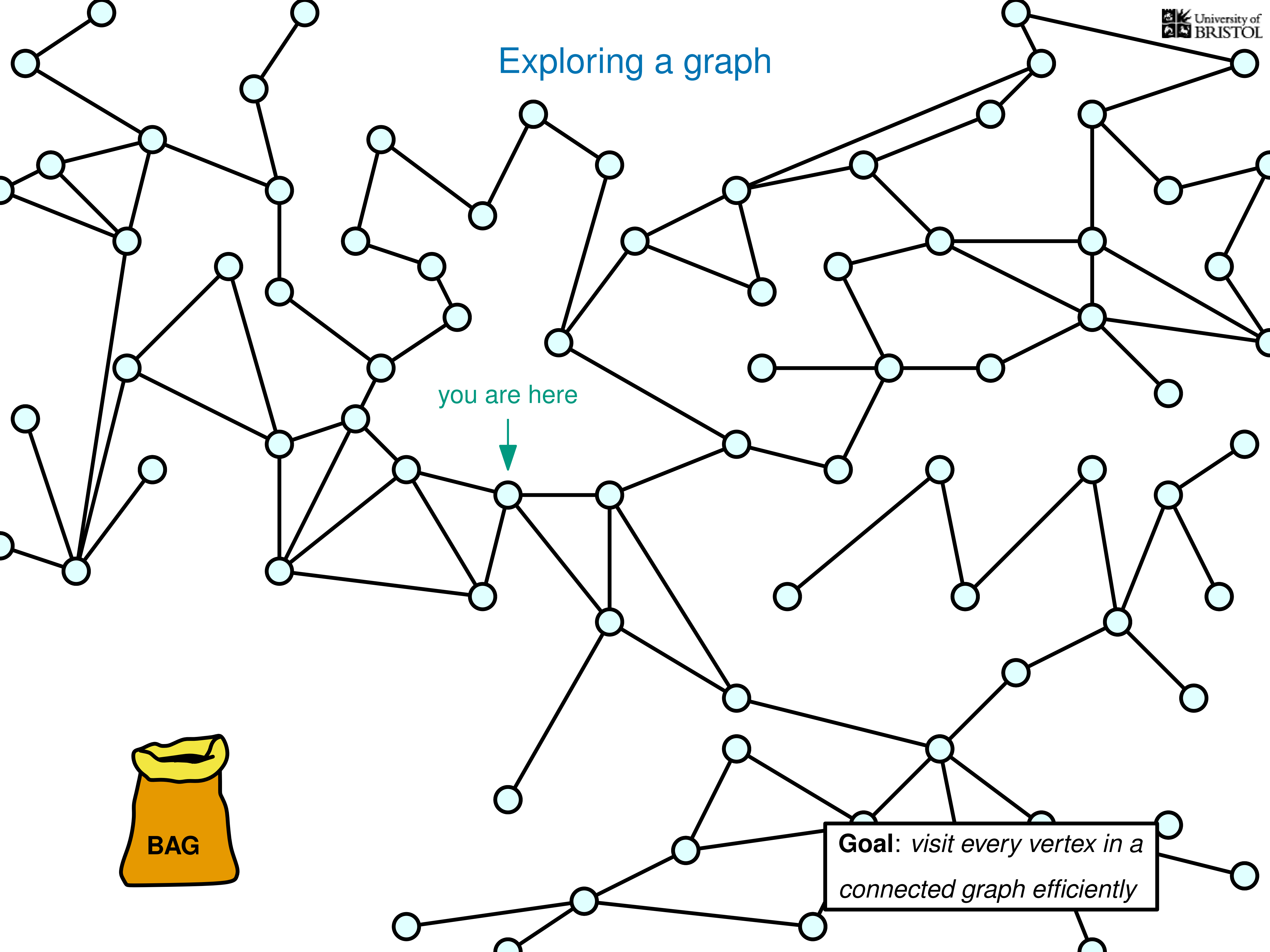
there is an edge if you can get from

one state to another in a single move

Exploring a graph

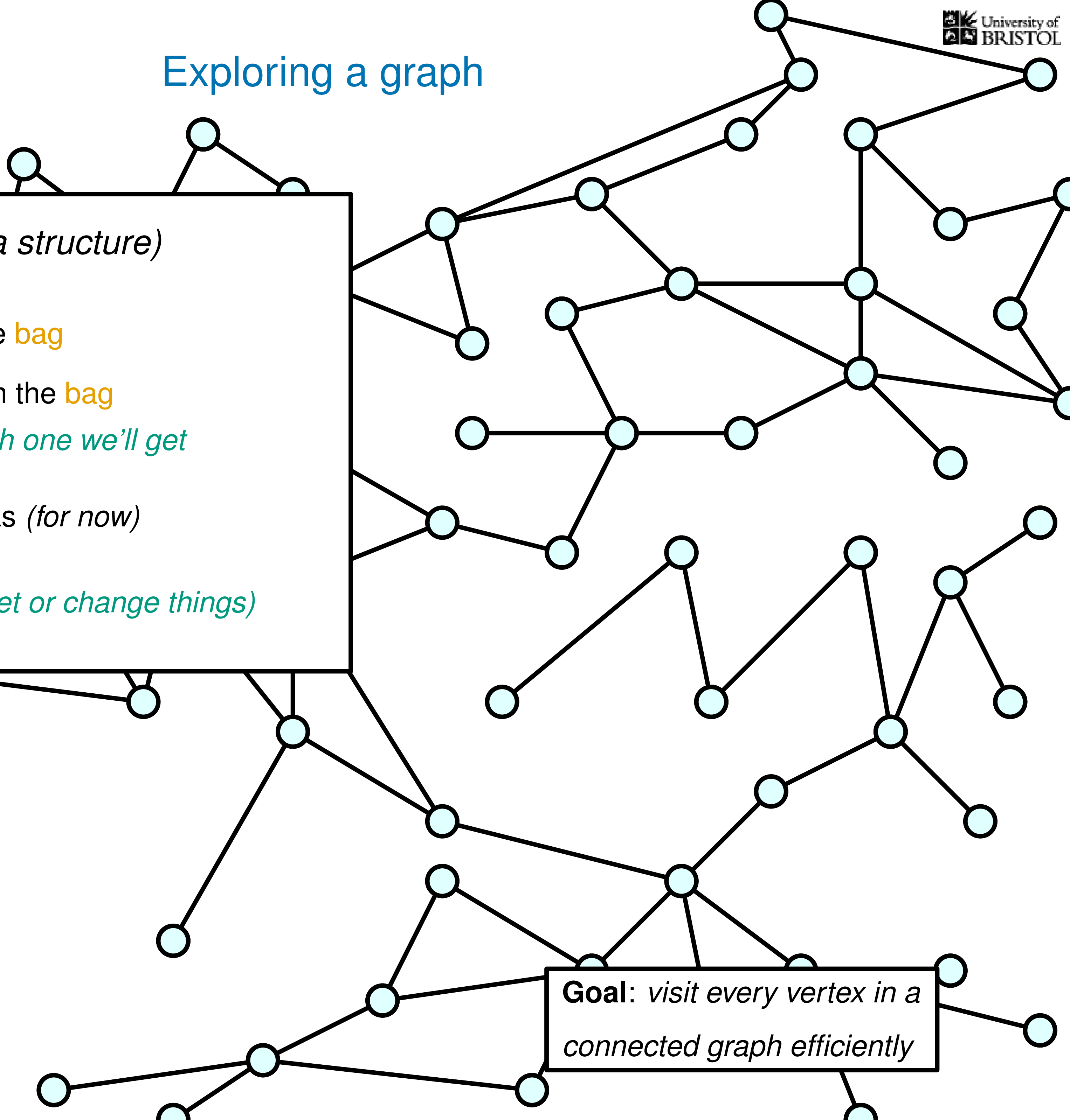


Exploring a graph



Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph



The bag *(secretly a data structure)*

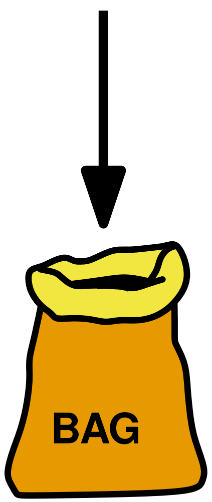
We can put a vertex in the bag

We can take a vertex from the bag

but we don't know which one we'll get

We don't care how it works *(for now)*

(the bag doesn't forget or change things)



Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

The bag *(secretly a data structure)*

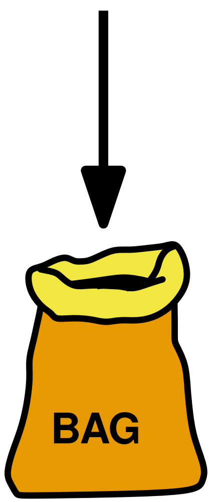
We can put a vertex in the bag

We can take a vertex from the bag

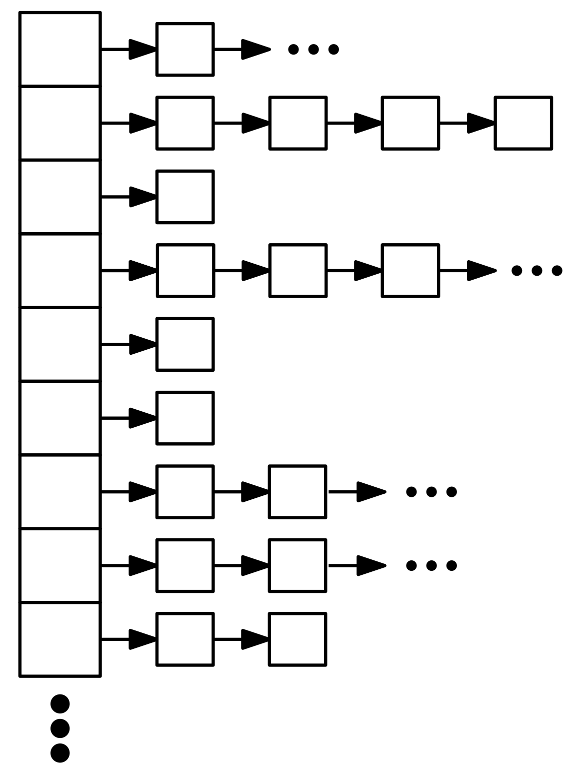
but we don't know which one we'll get

We don't care how it works *(for now)*

(the bag doesn't forget or change things)

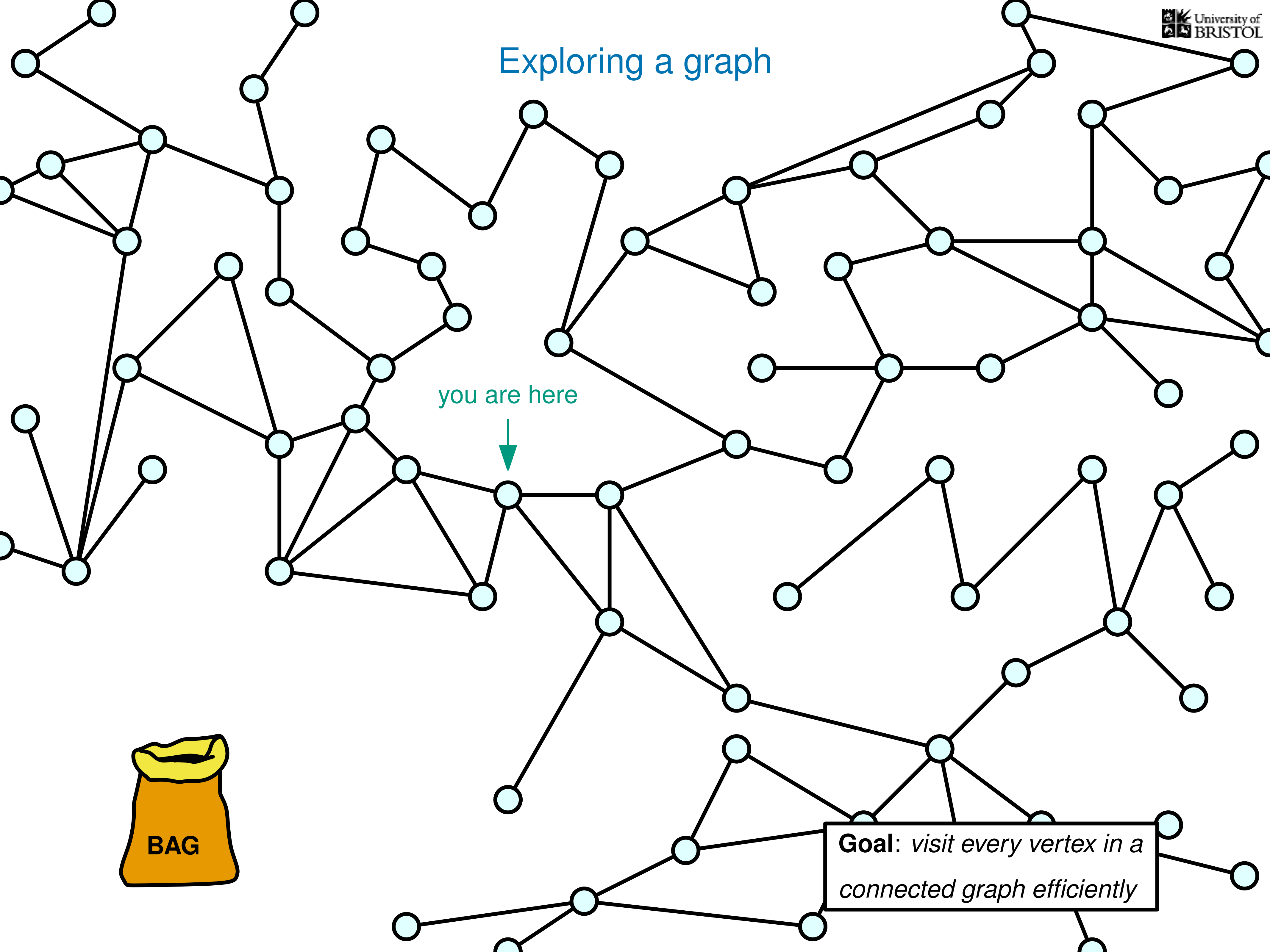


We going to use an adjacency list so the graph is really stored like this:



Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

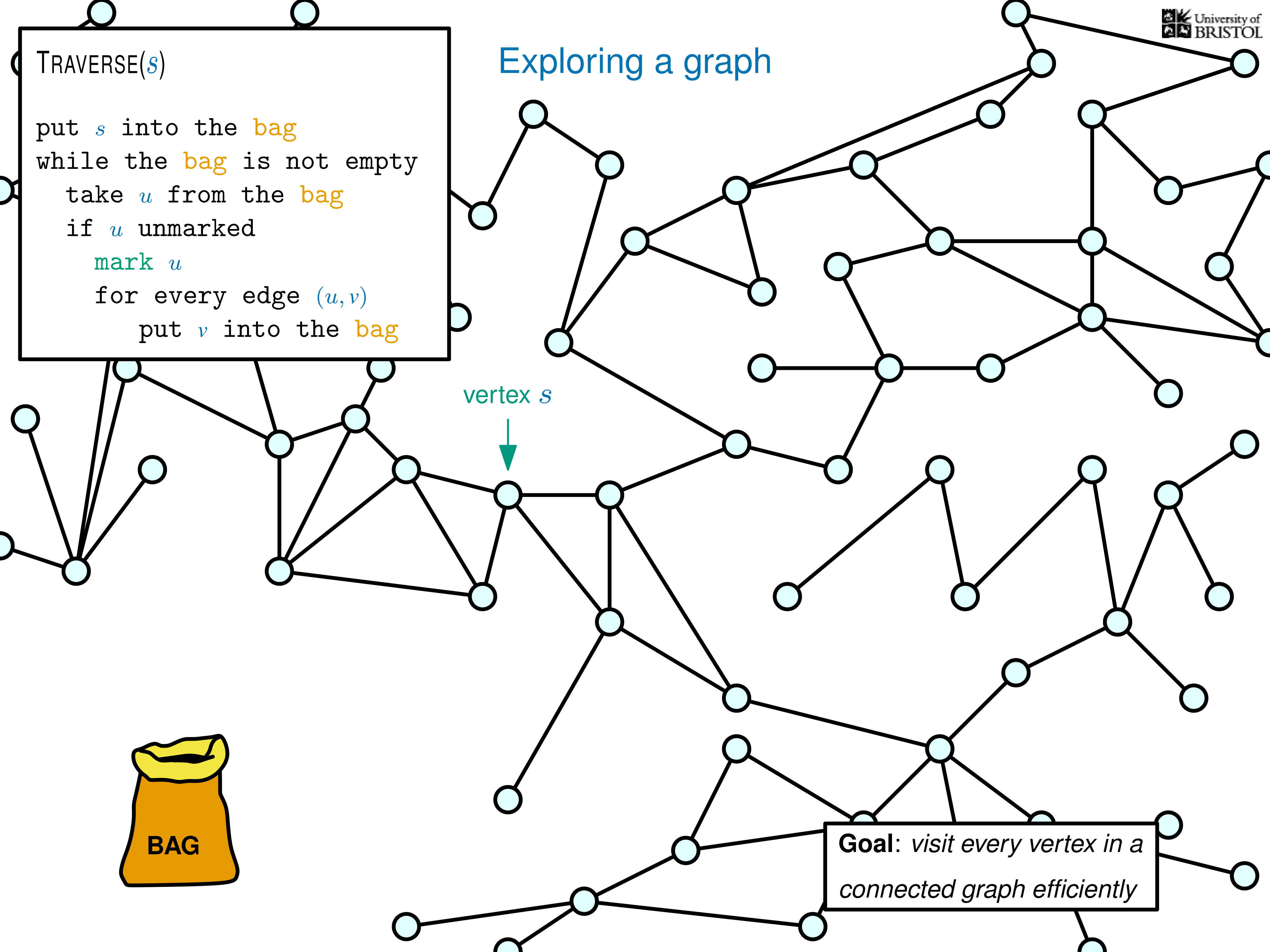


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

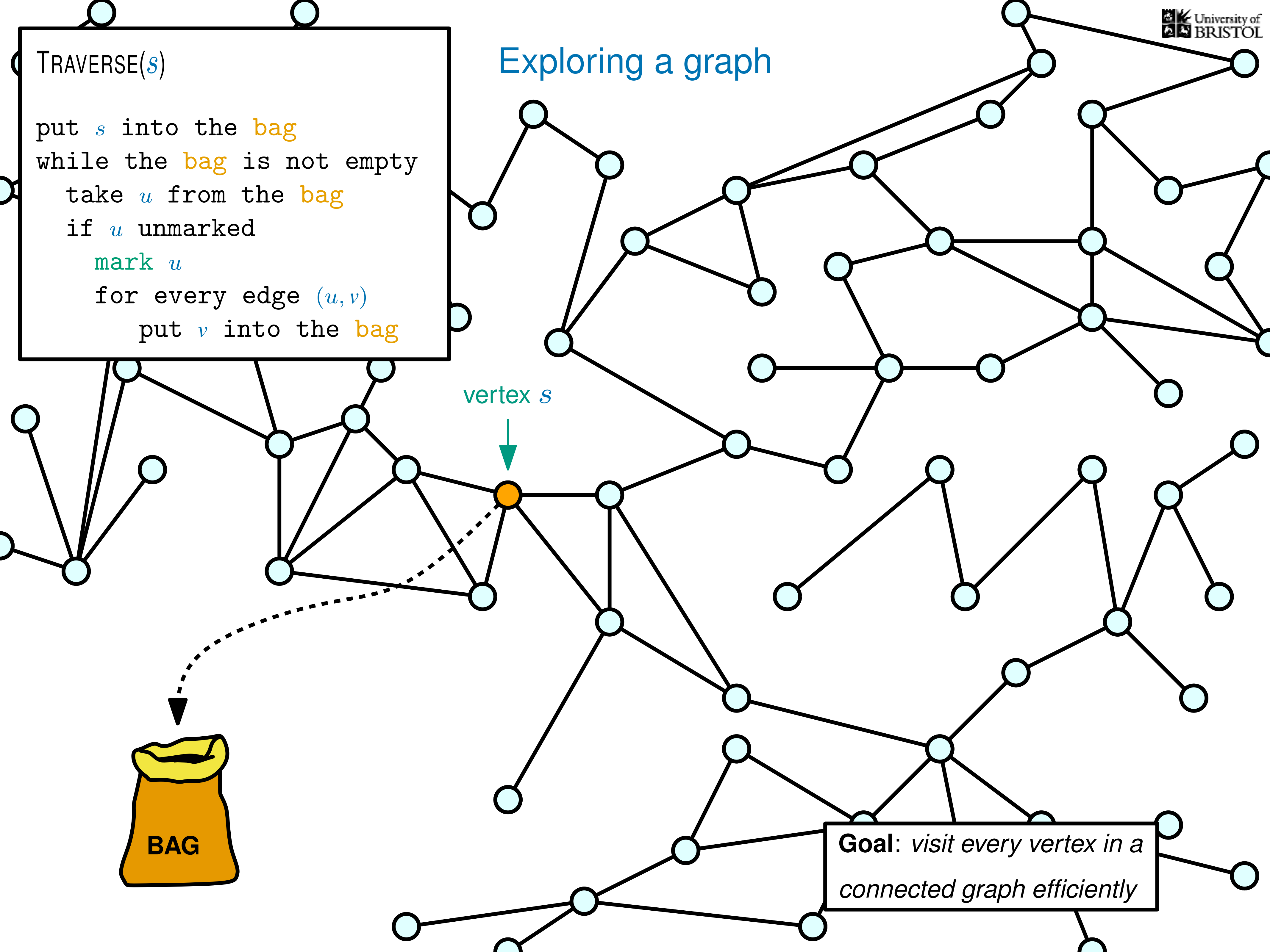


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

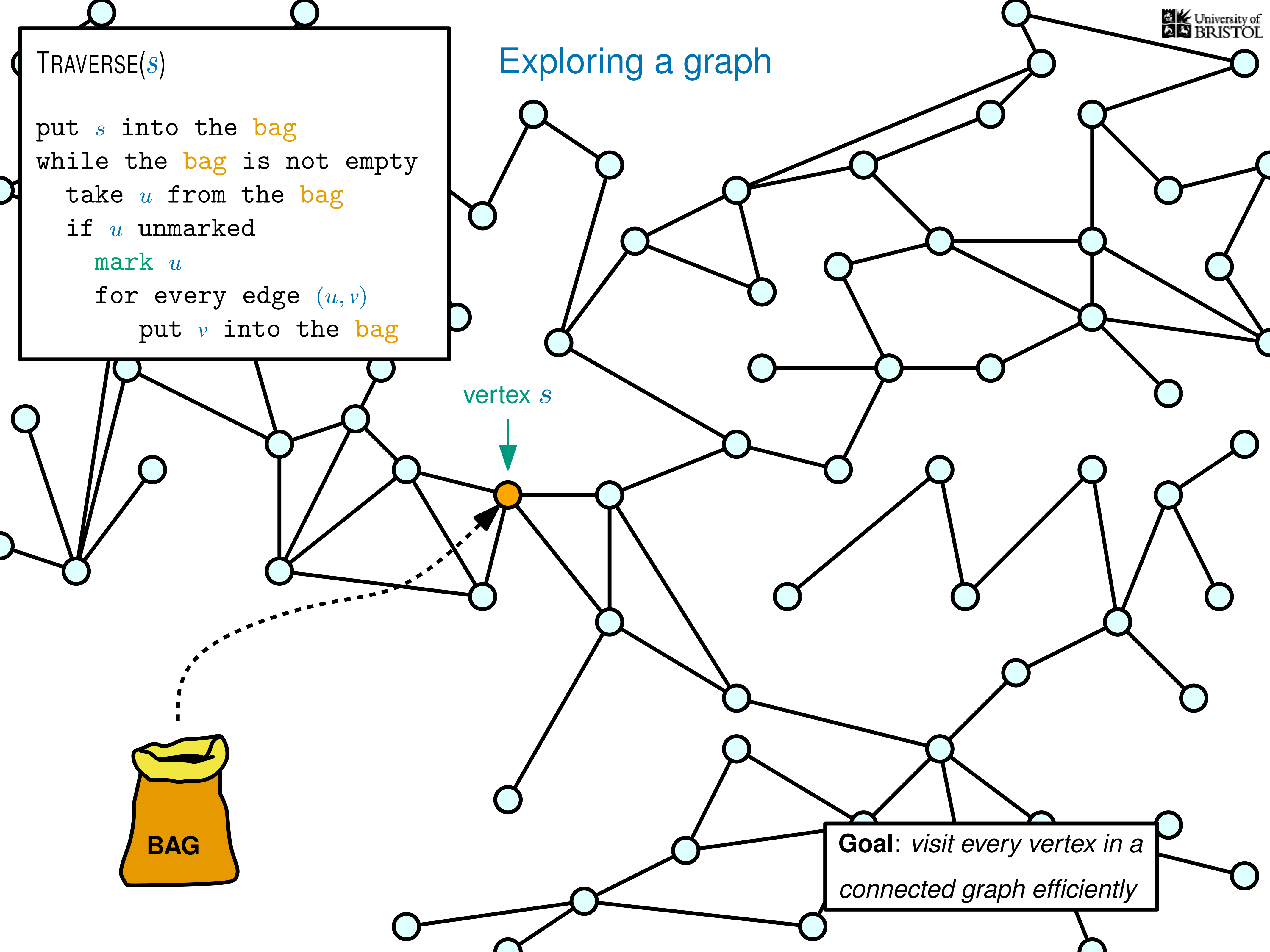


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

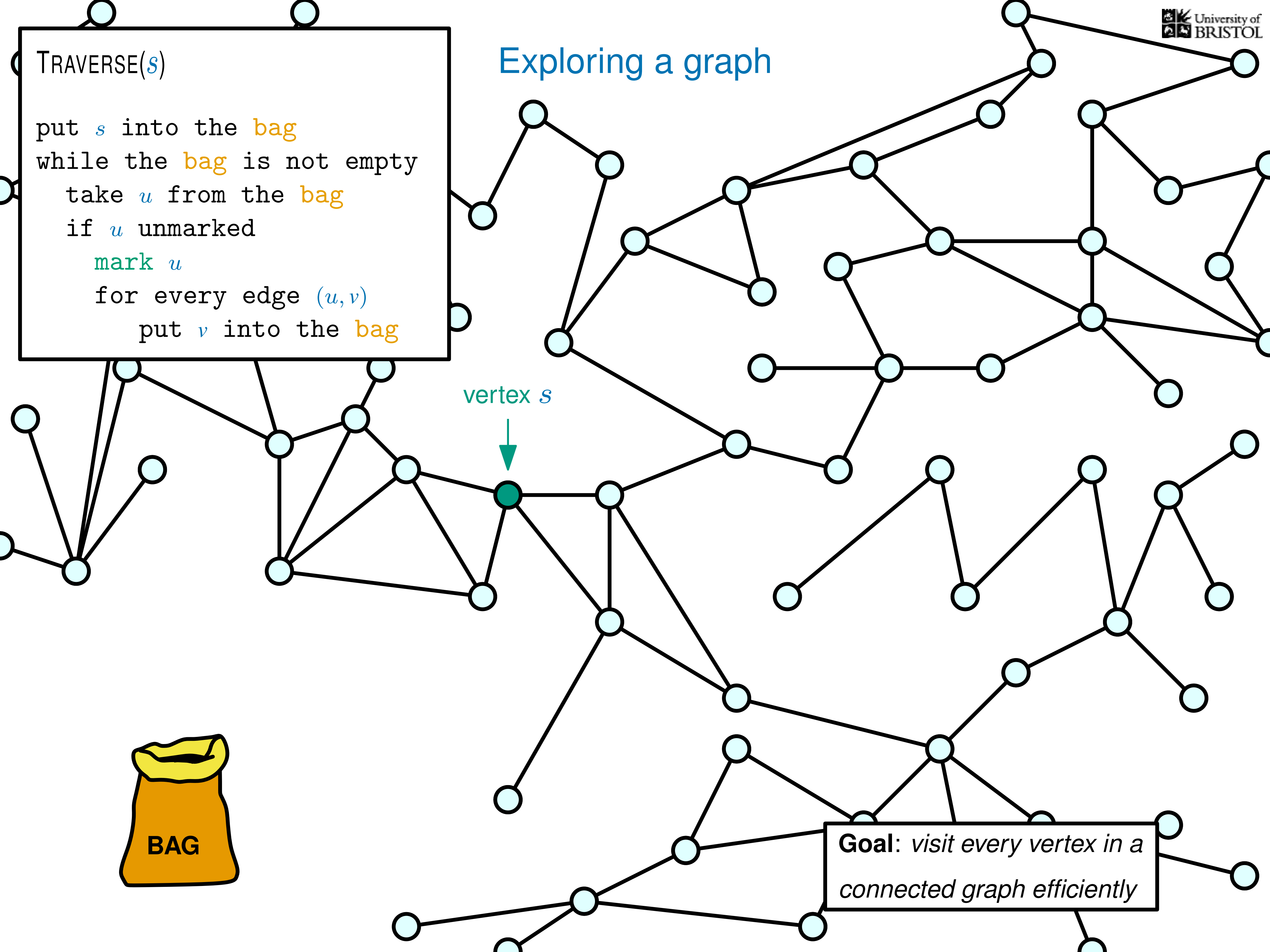


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

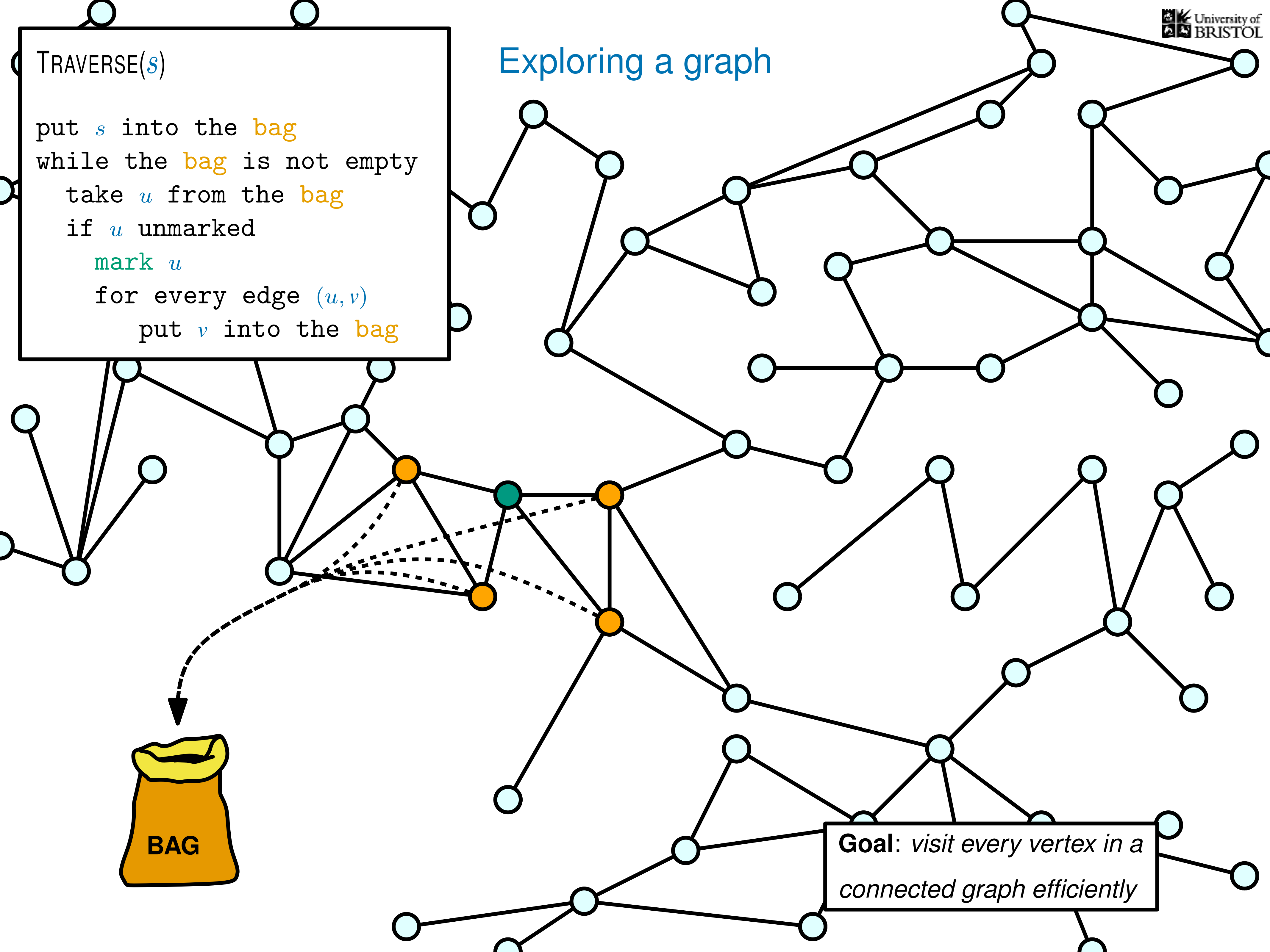


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

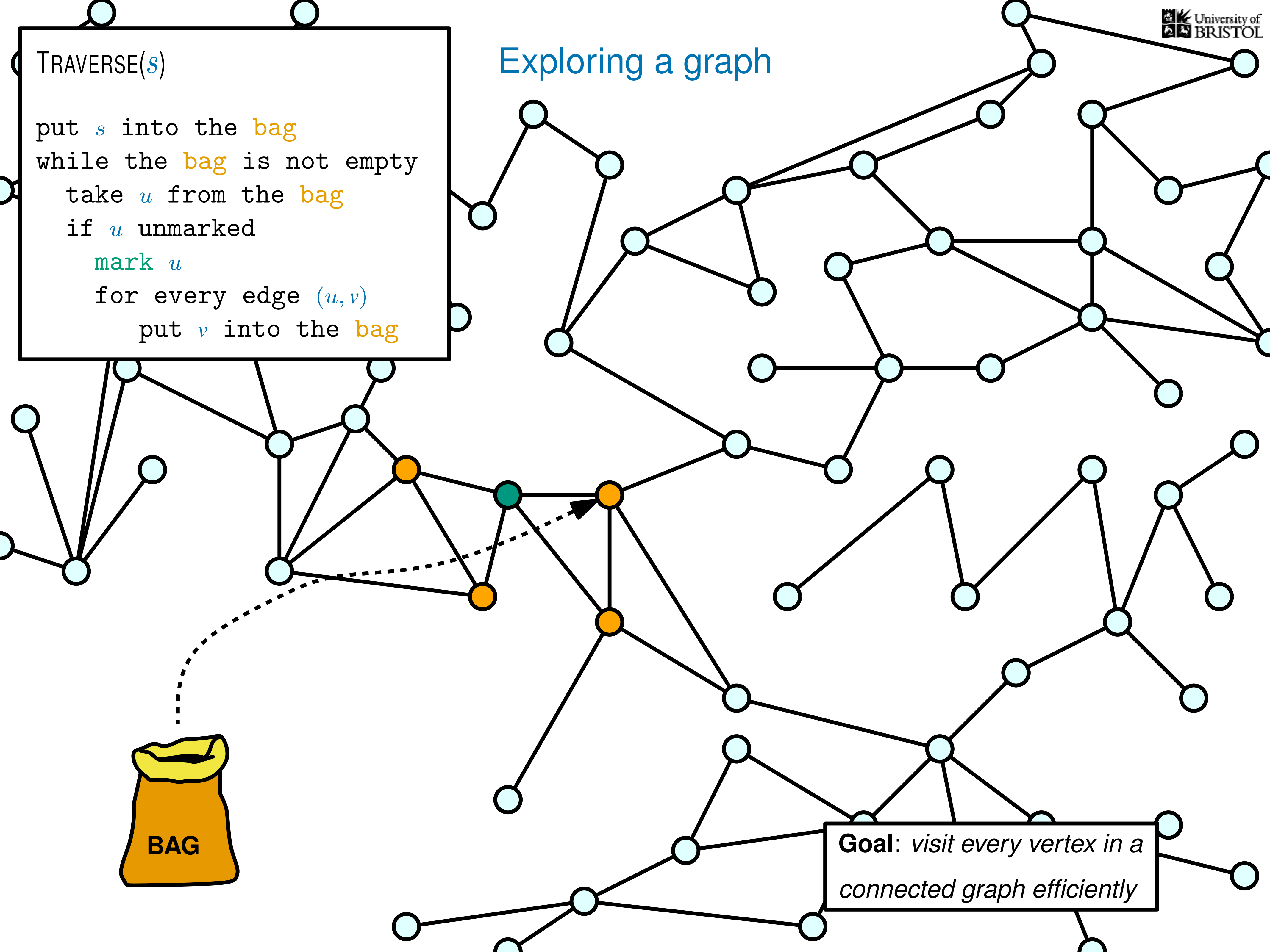


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
      put v into the bag
```

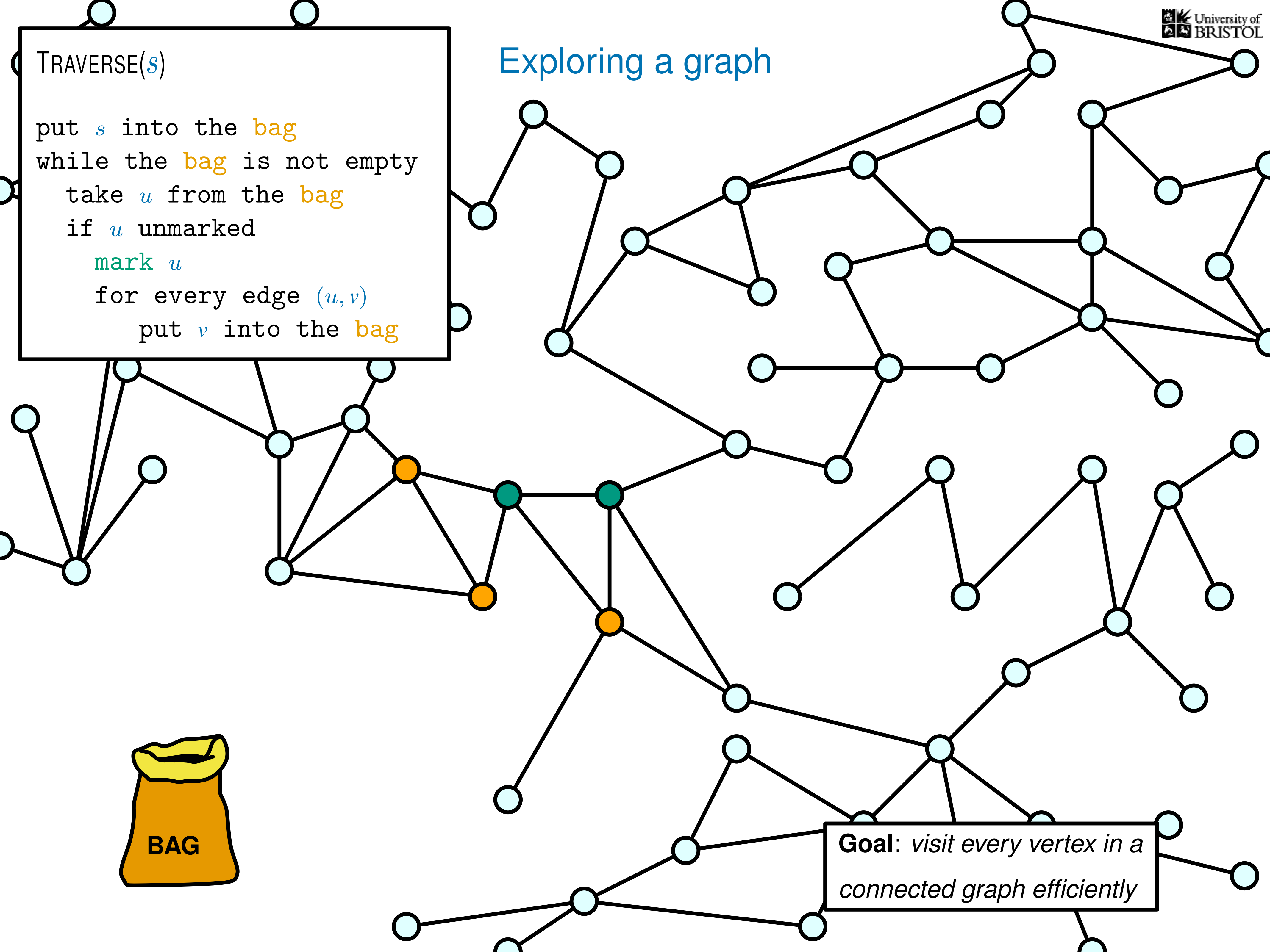


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
      put v into the bag
```

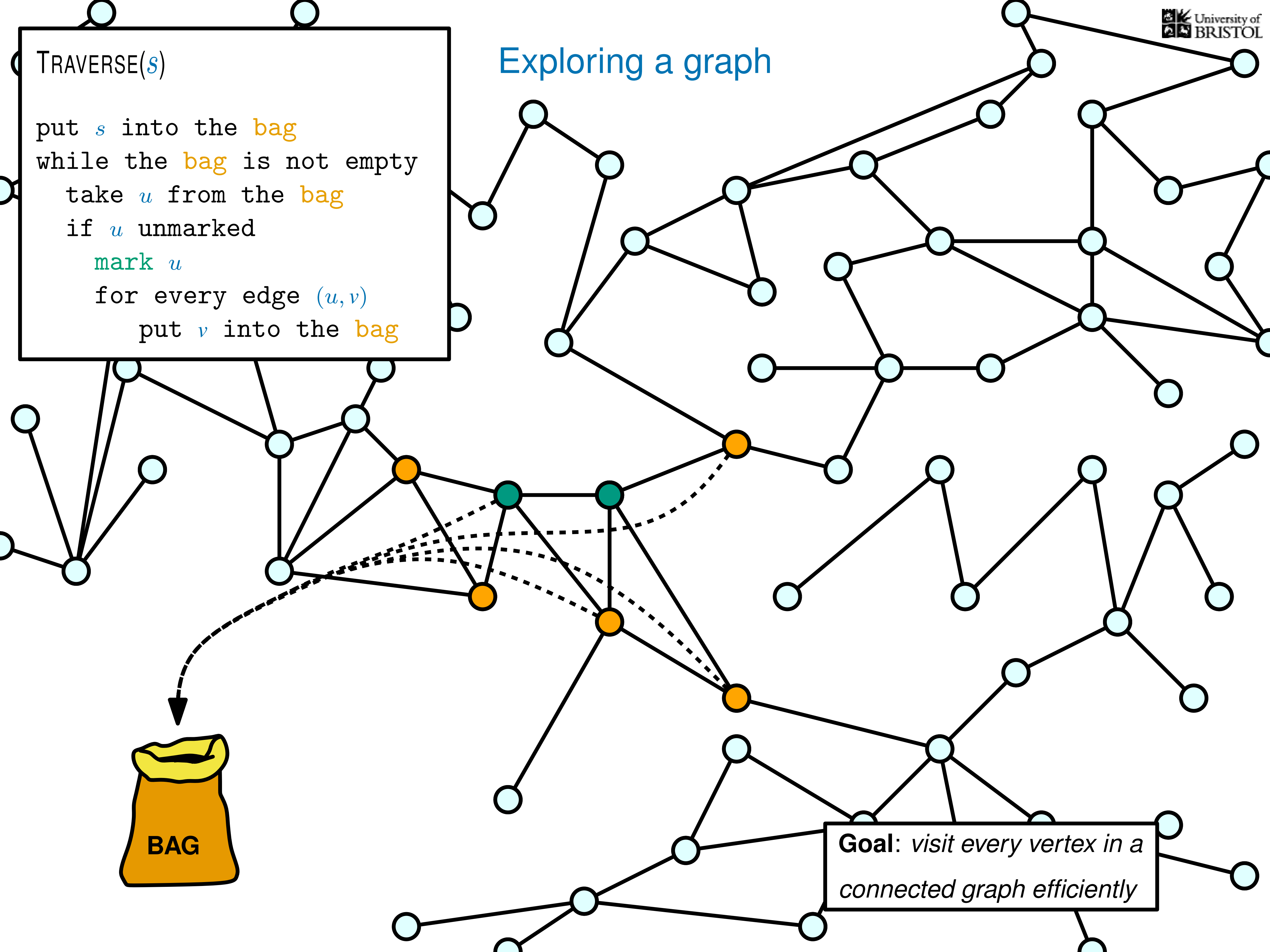


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph


```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

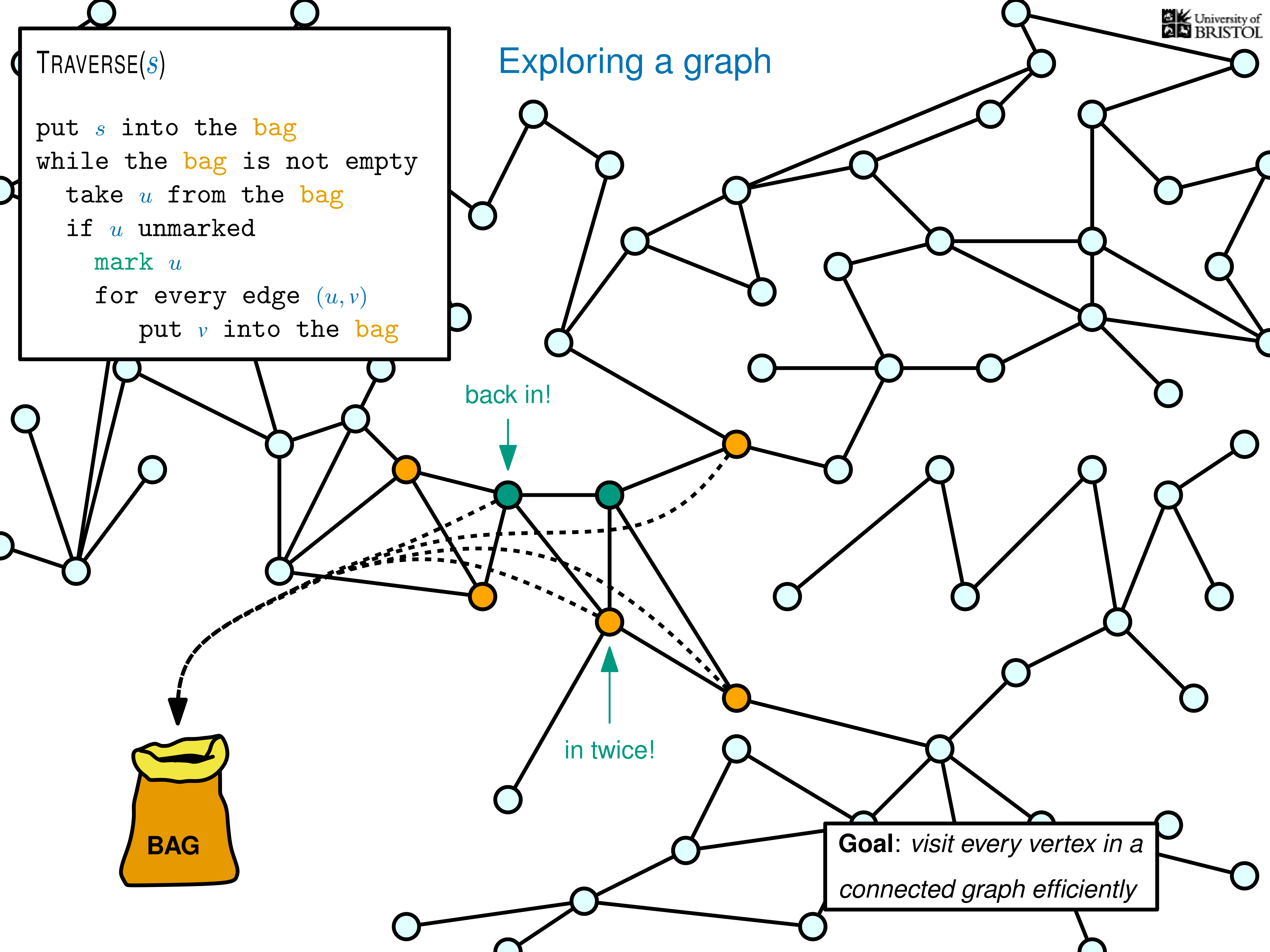


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

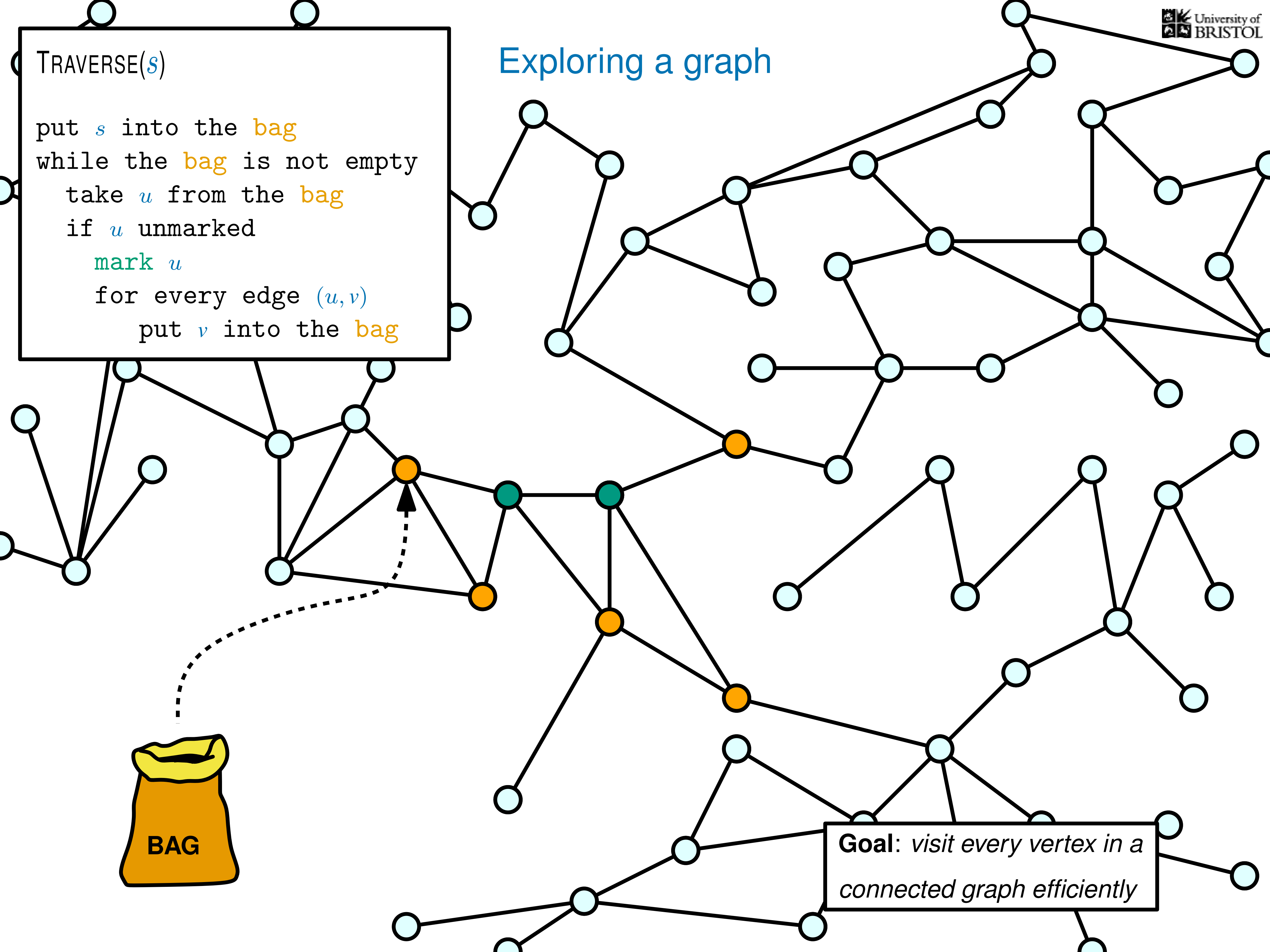


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

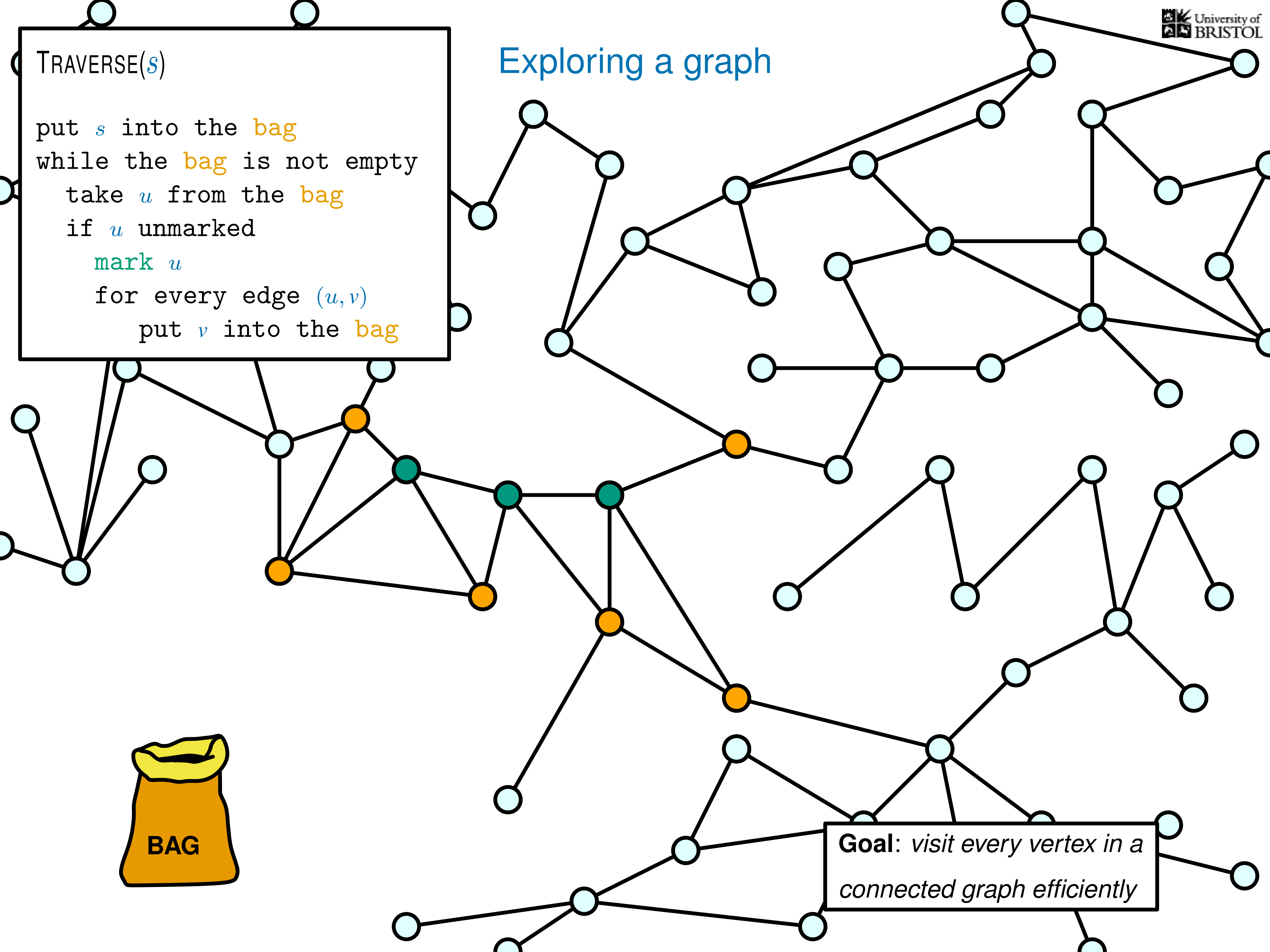


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

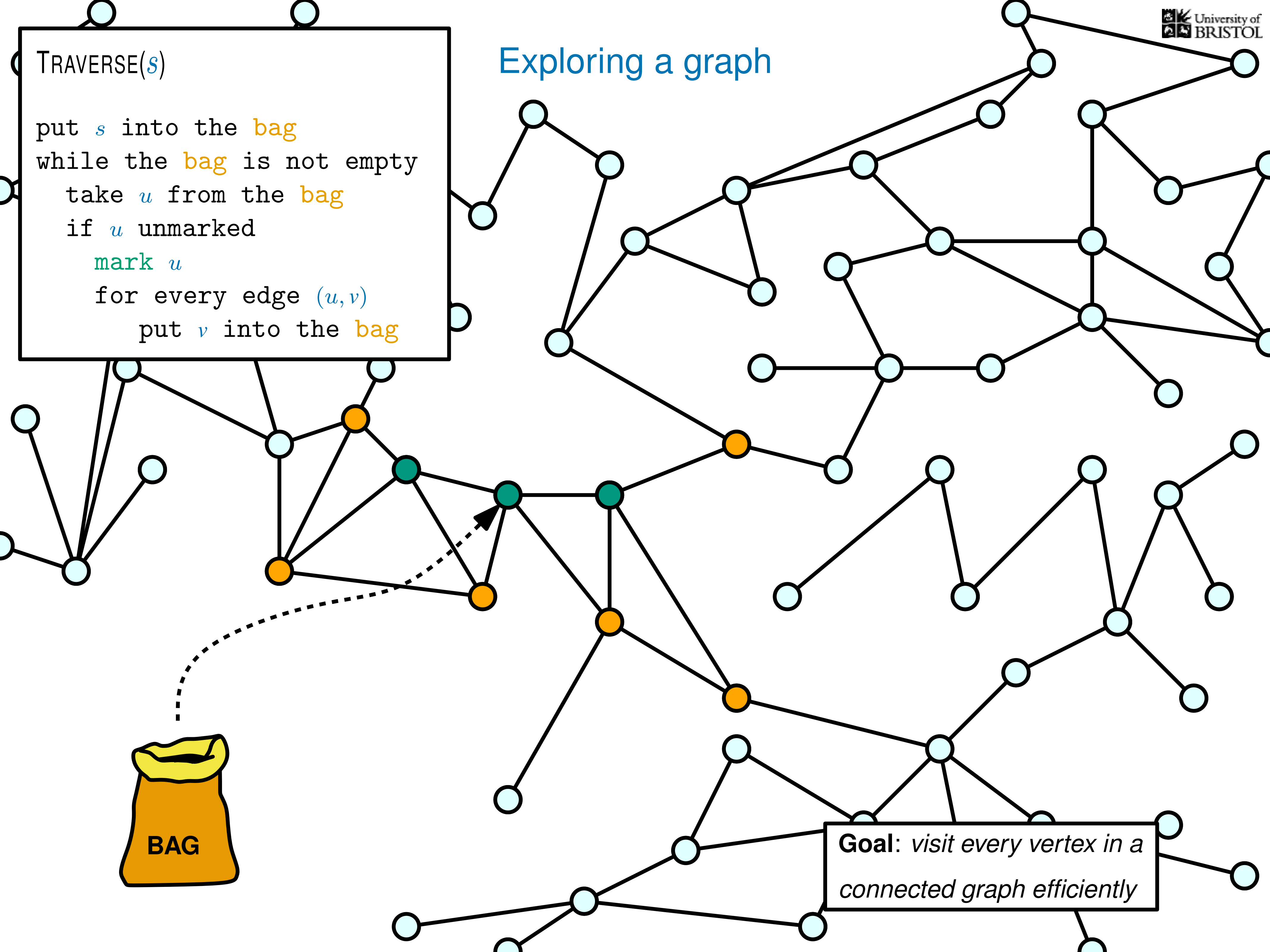


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

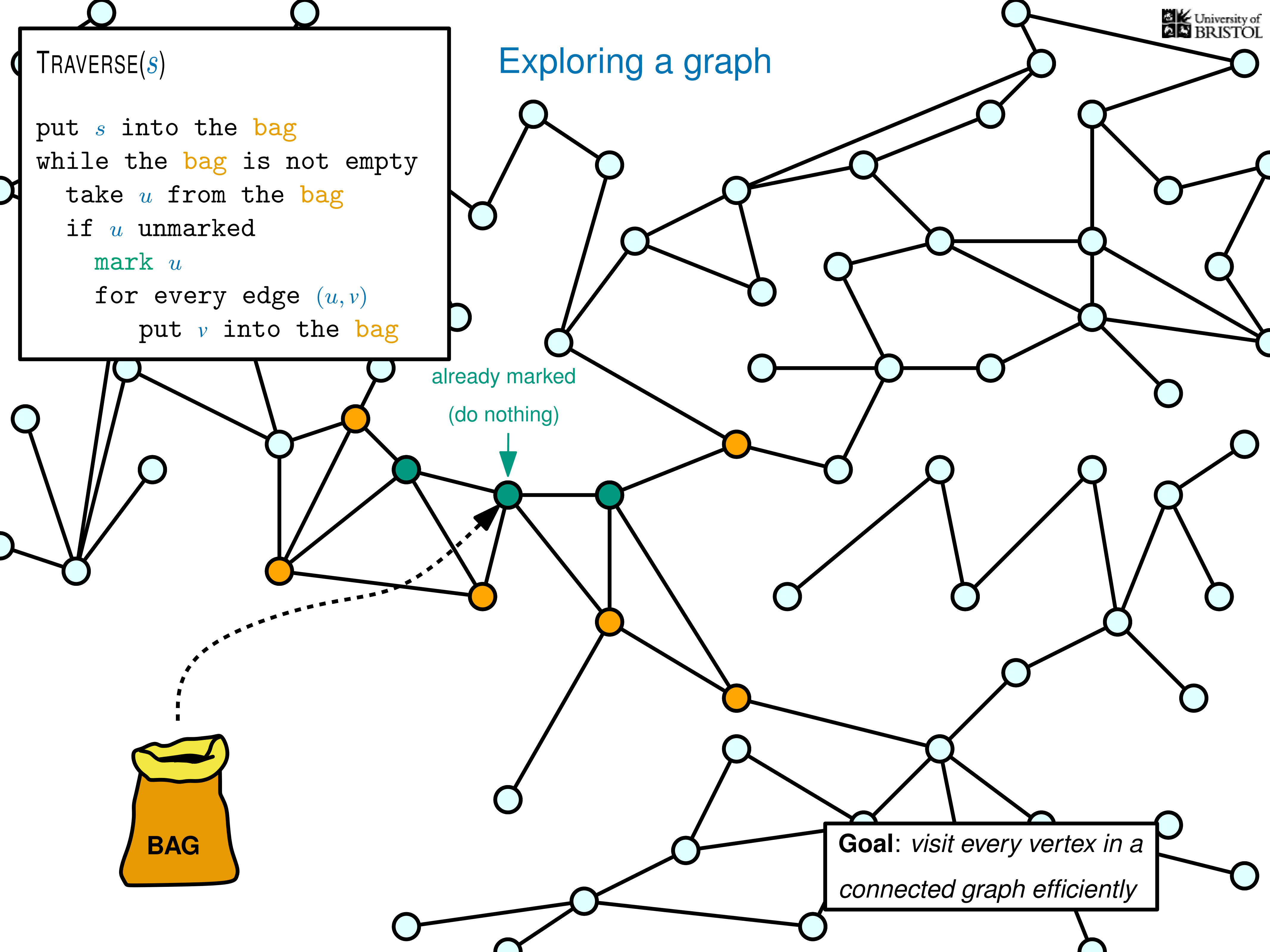


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

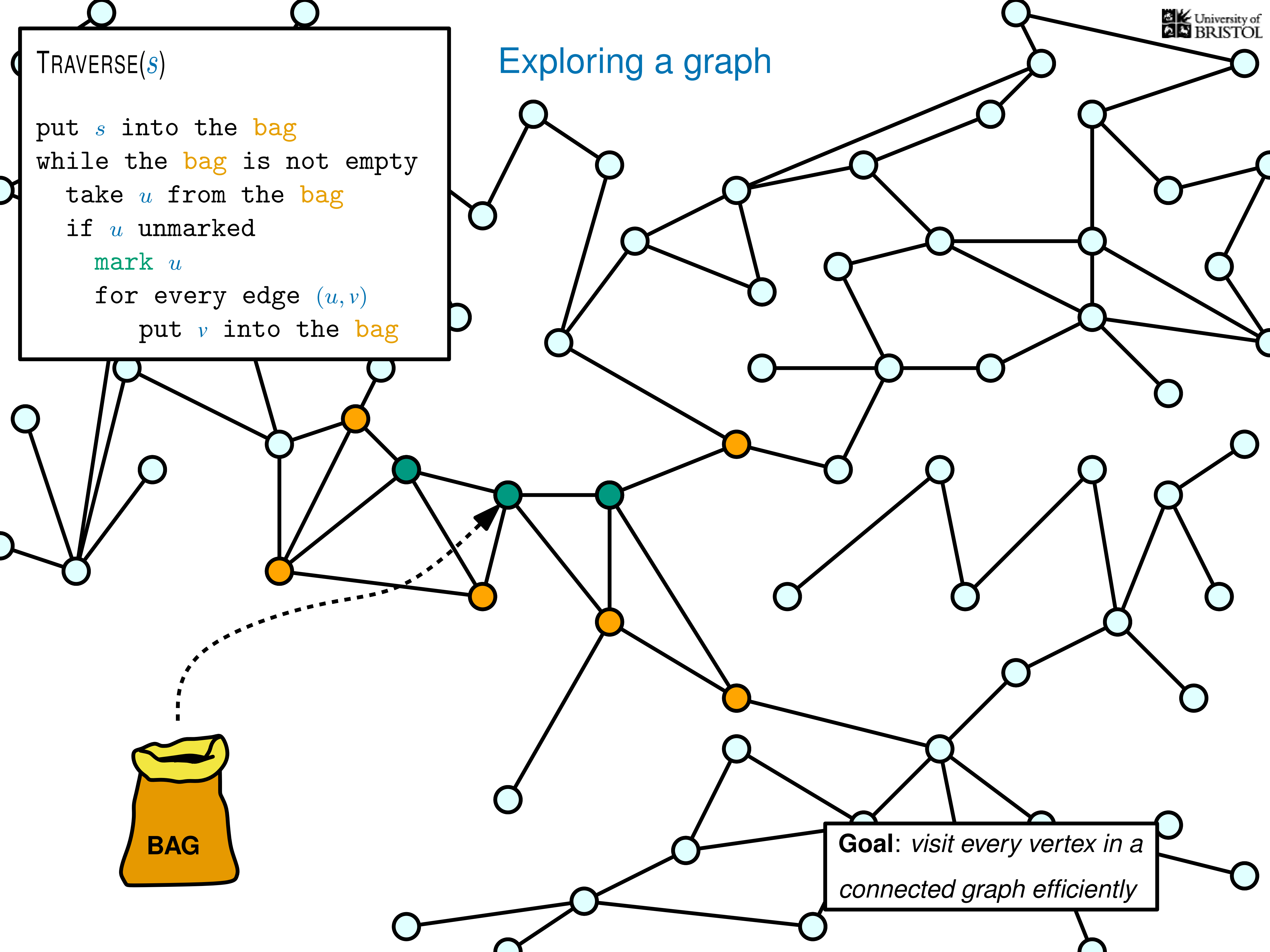


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

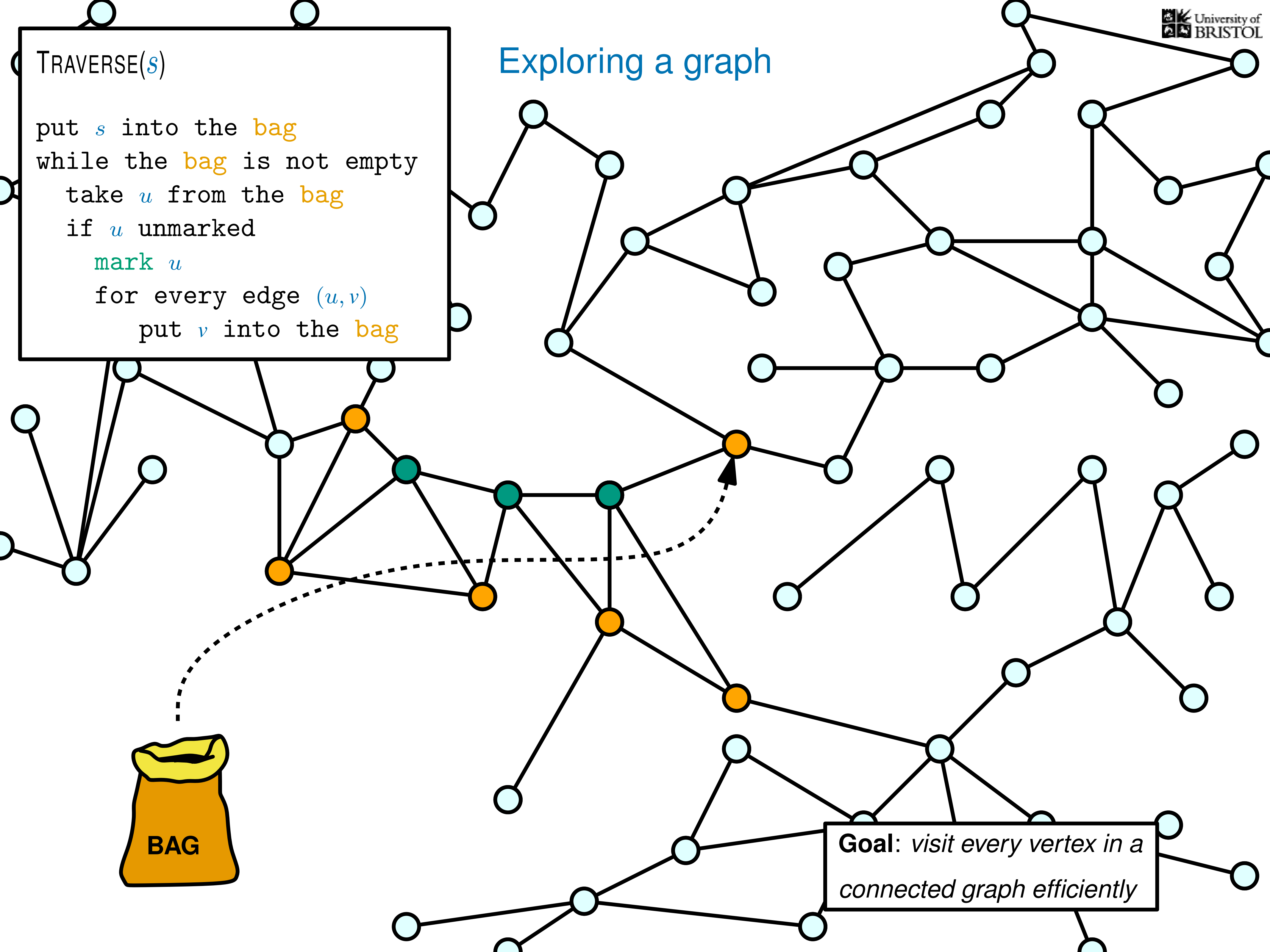


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

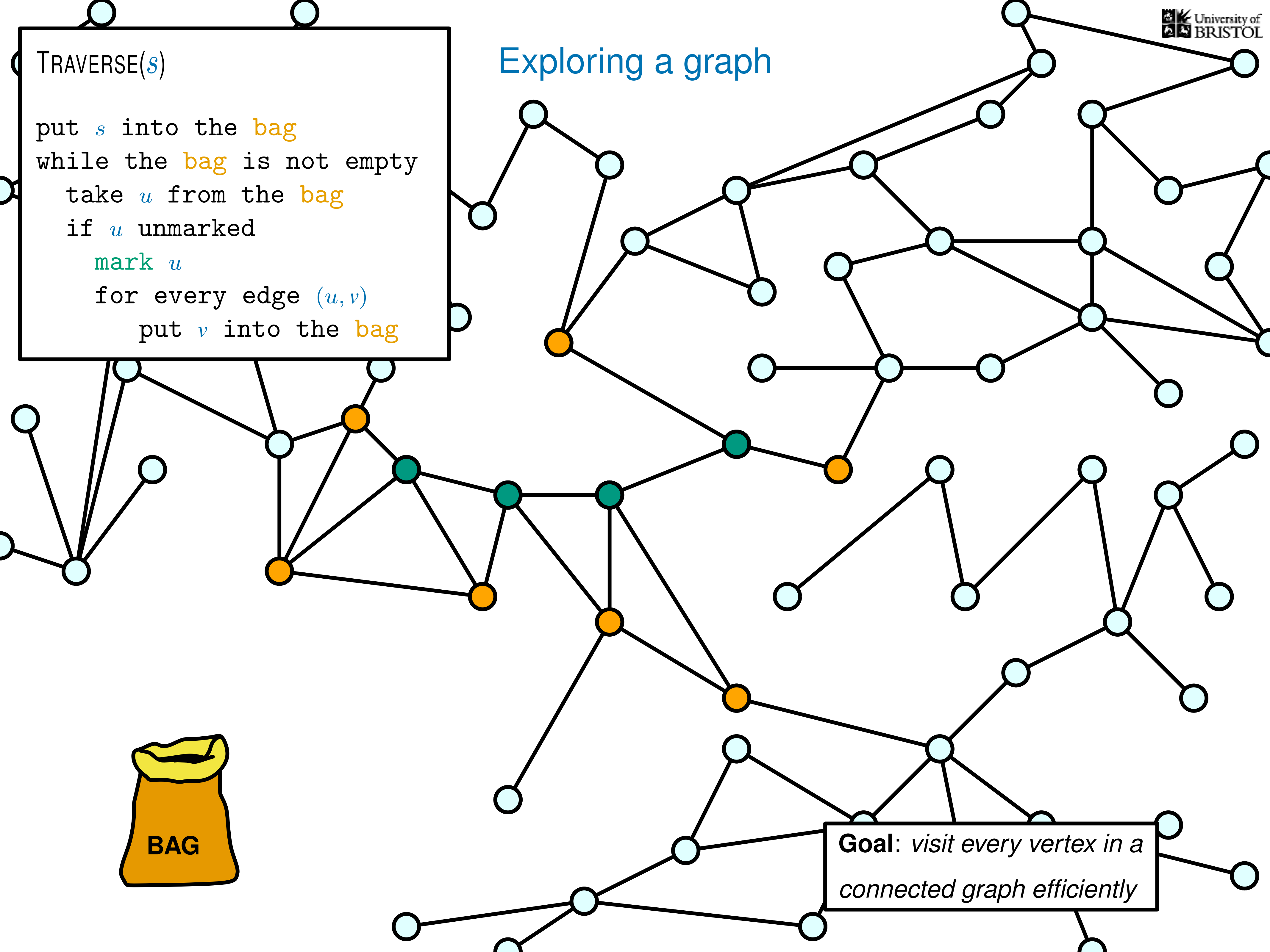


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
      put v into the bag
```



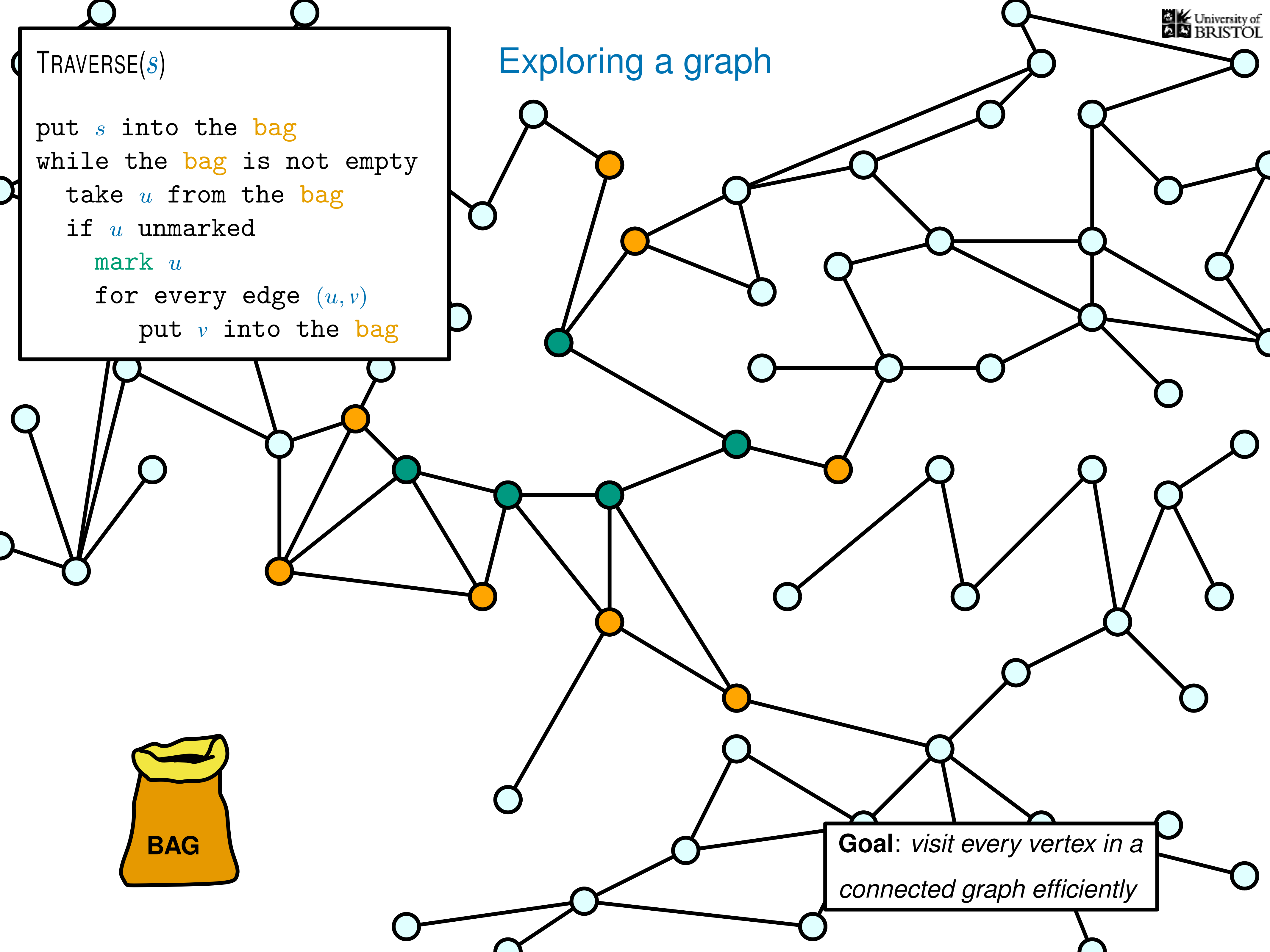
Goal: *visit every vertex in a connected graph efficiently*

University of BRISTOL

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```



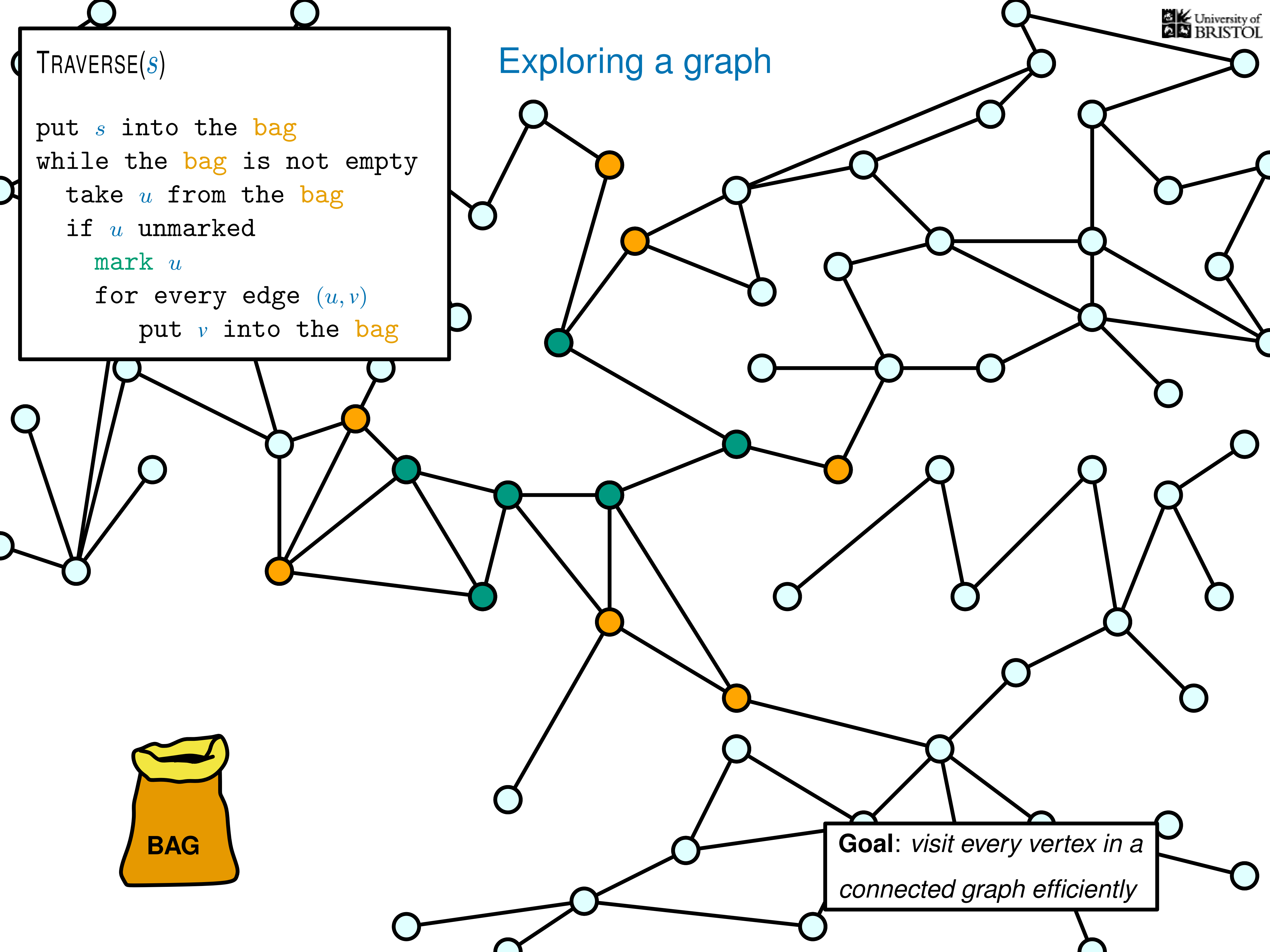
Goal: *visit every vertex in a connected graph efficiently*

University of BRISTOL

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

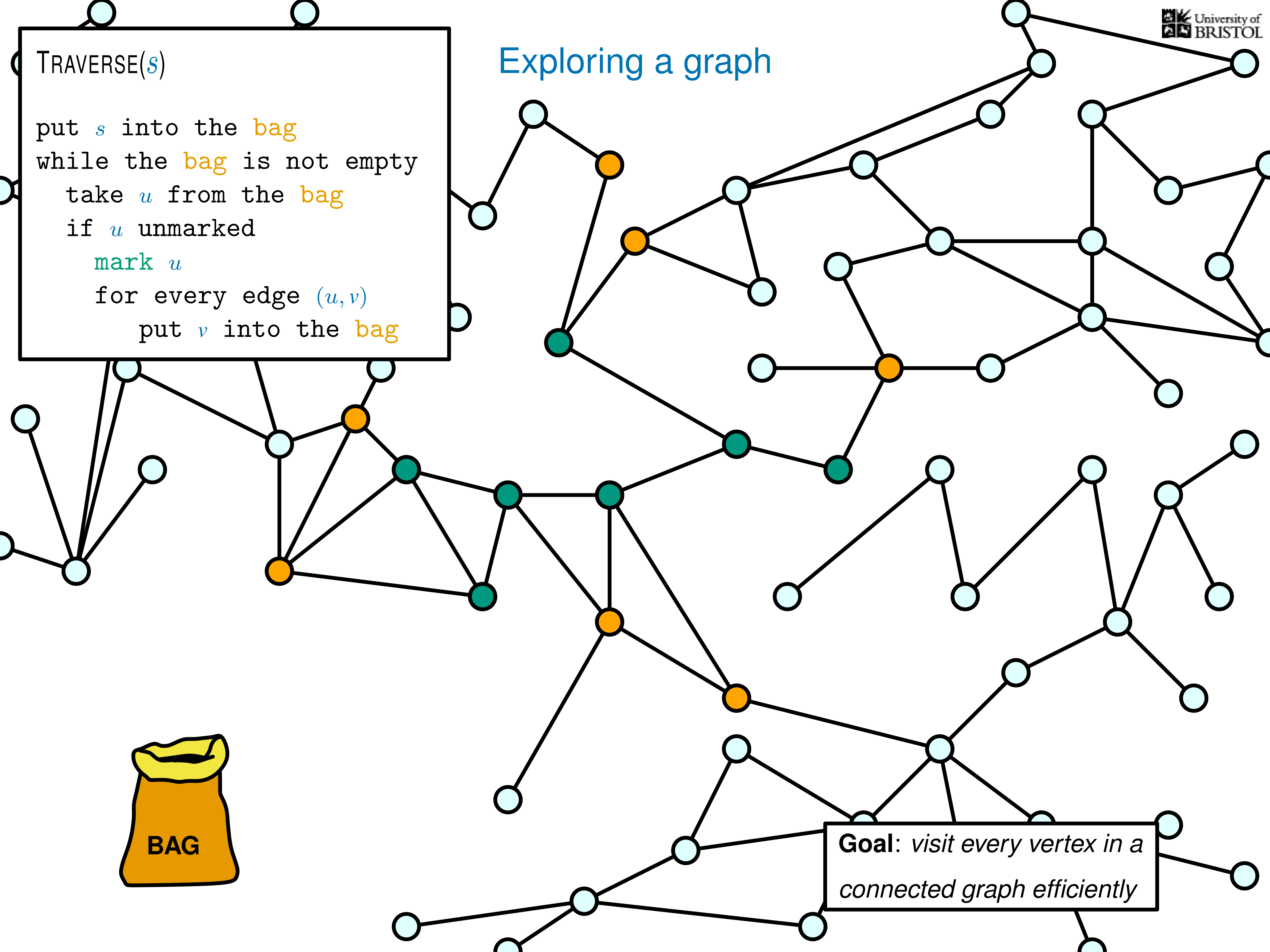


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

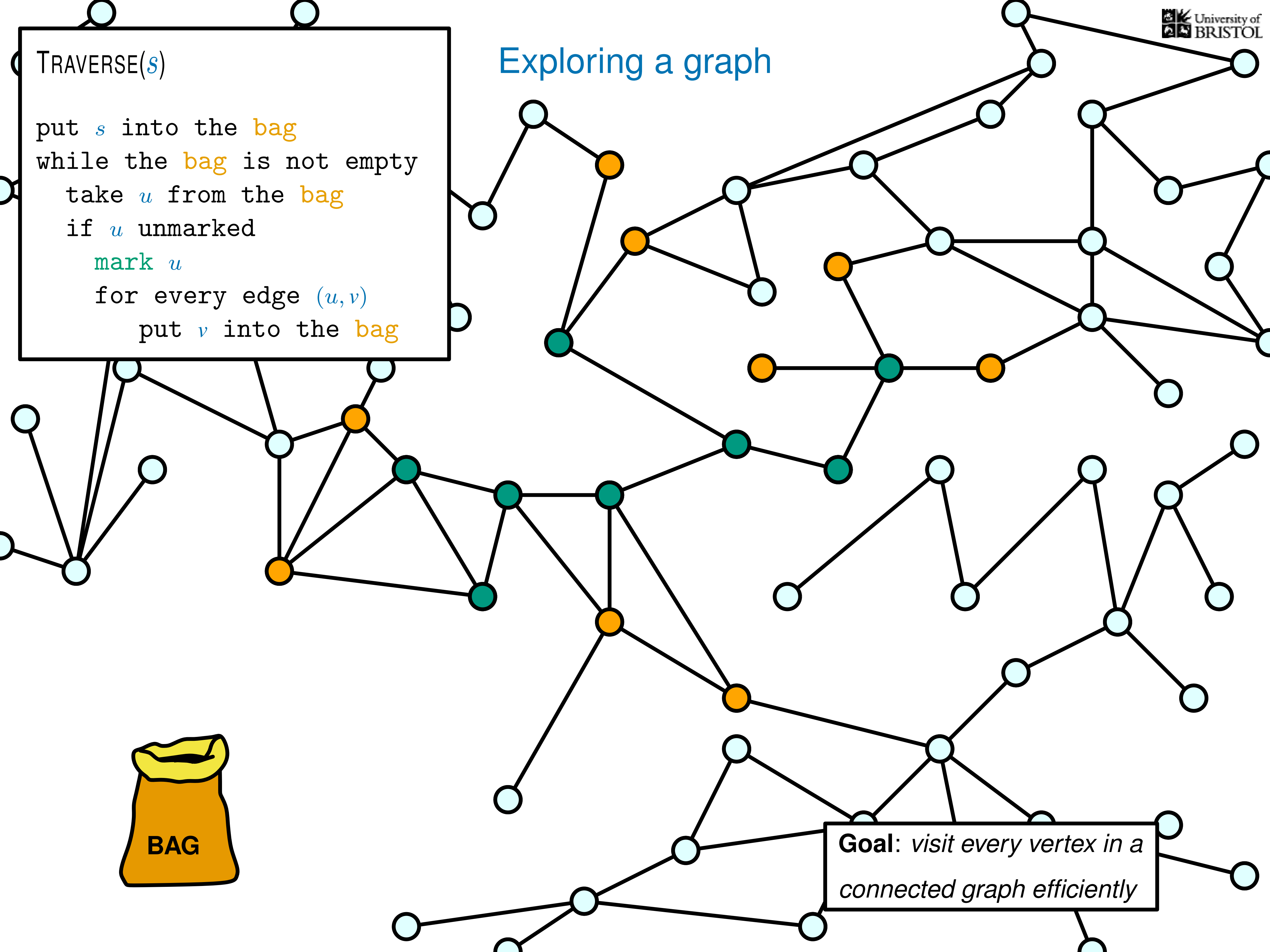


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

Traverse(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

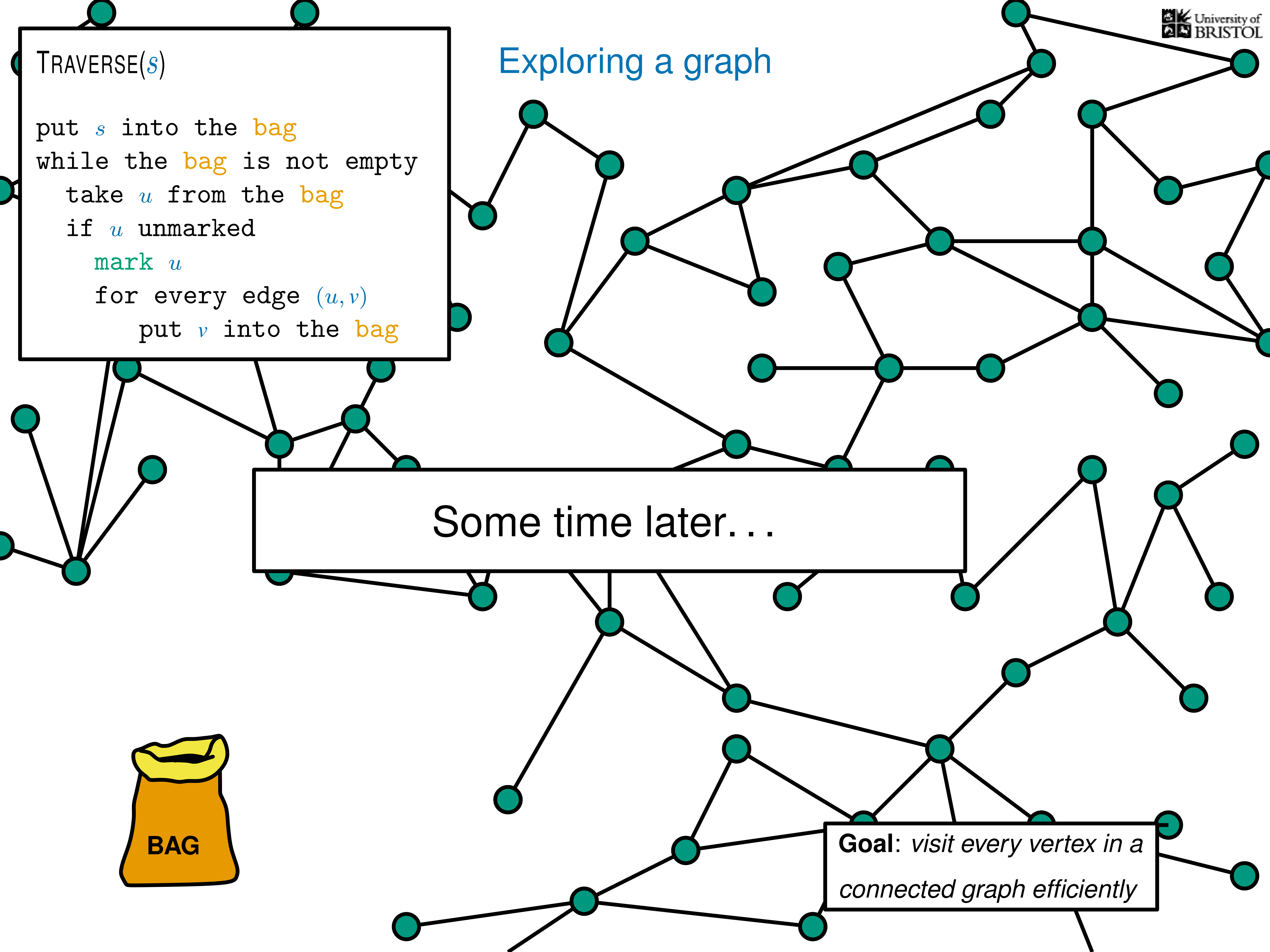


Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

```
Traverse(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```



Some time later...

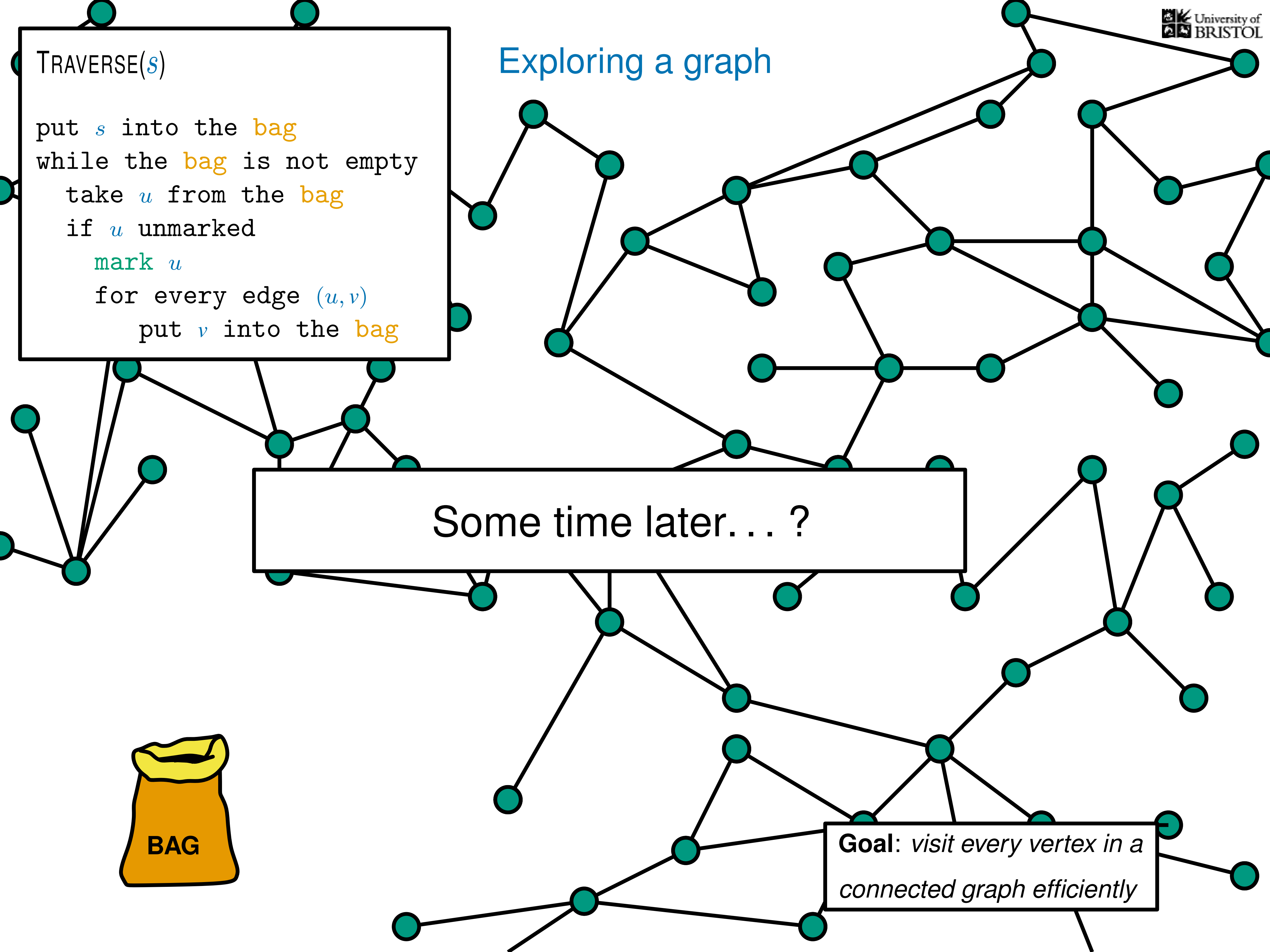
Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

Some time later. . . ?



Goal: *visit every vertex in a connected graph efficiently*

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

Questions:

1. *Does* TRAVERSE *always stop?*
2. *Does it always visit every vertex?*
3. *How fast is it? (in the worst case)*

Some time later. . . ?

Goal: *visit every vertex in a connected graph efficiently*

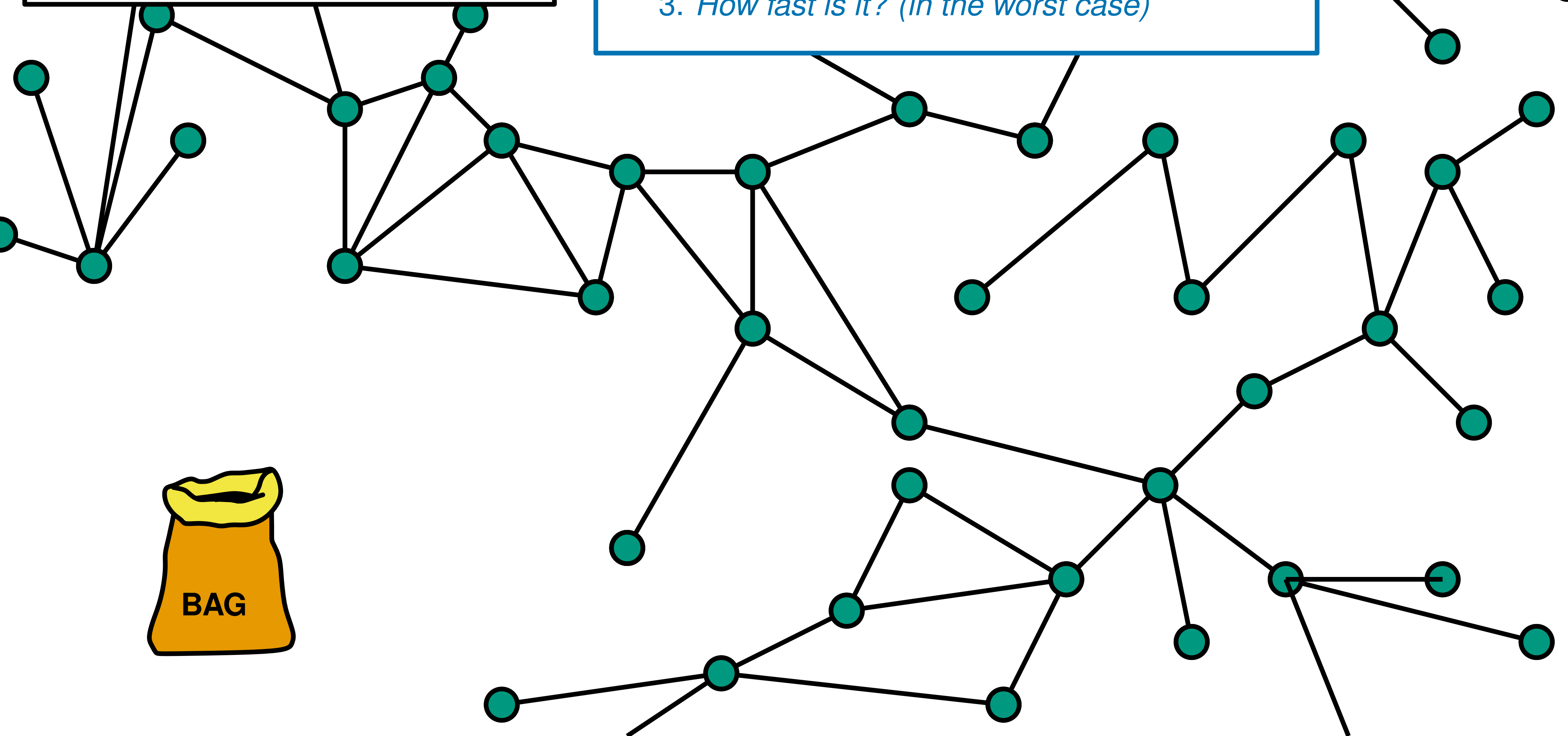
Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

Questions:

1. *Does* TRAVERSE *always stop?*
2. *Does it always visit every vertex?*
3. *How fast is it? (in the worst case)*

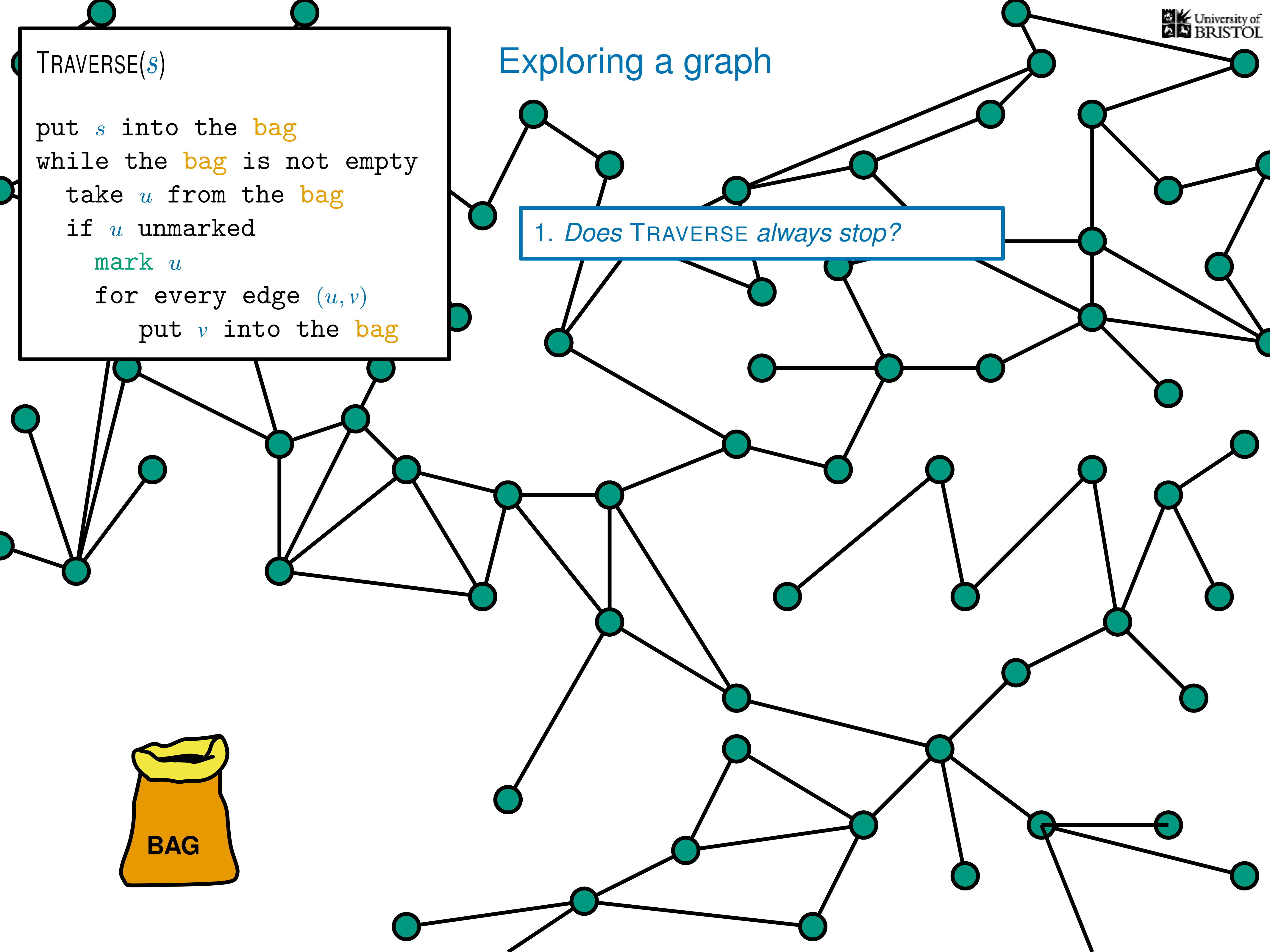


Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
      put v into the bag
```

1. *Does* TRAVERSE *always stop?*

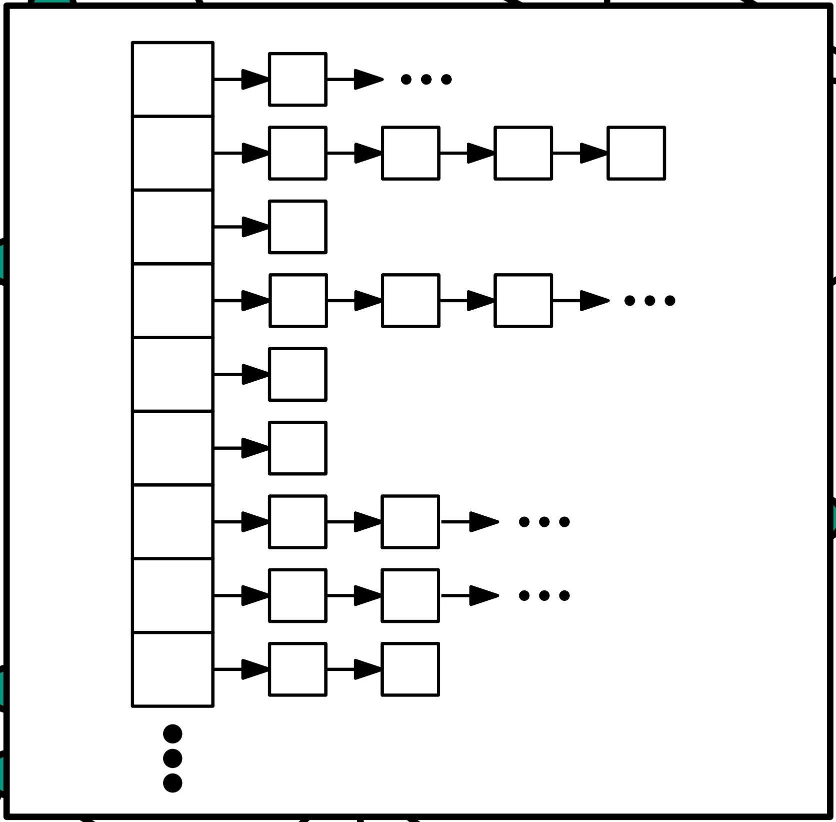


Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

1. *Does* TRAVERSE *always stop?*

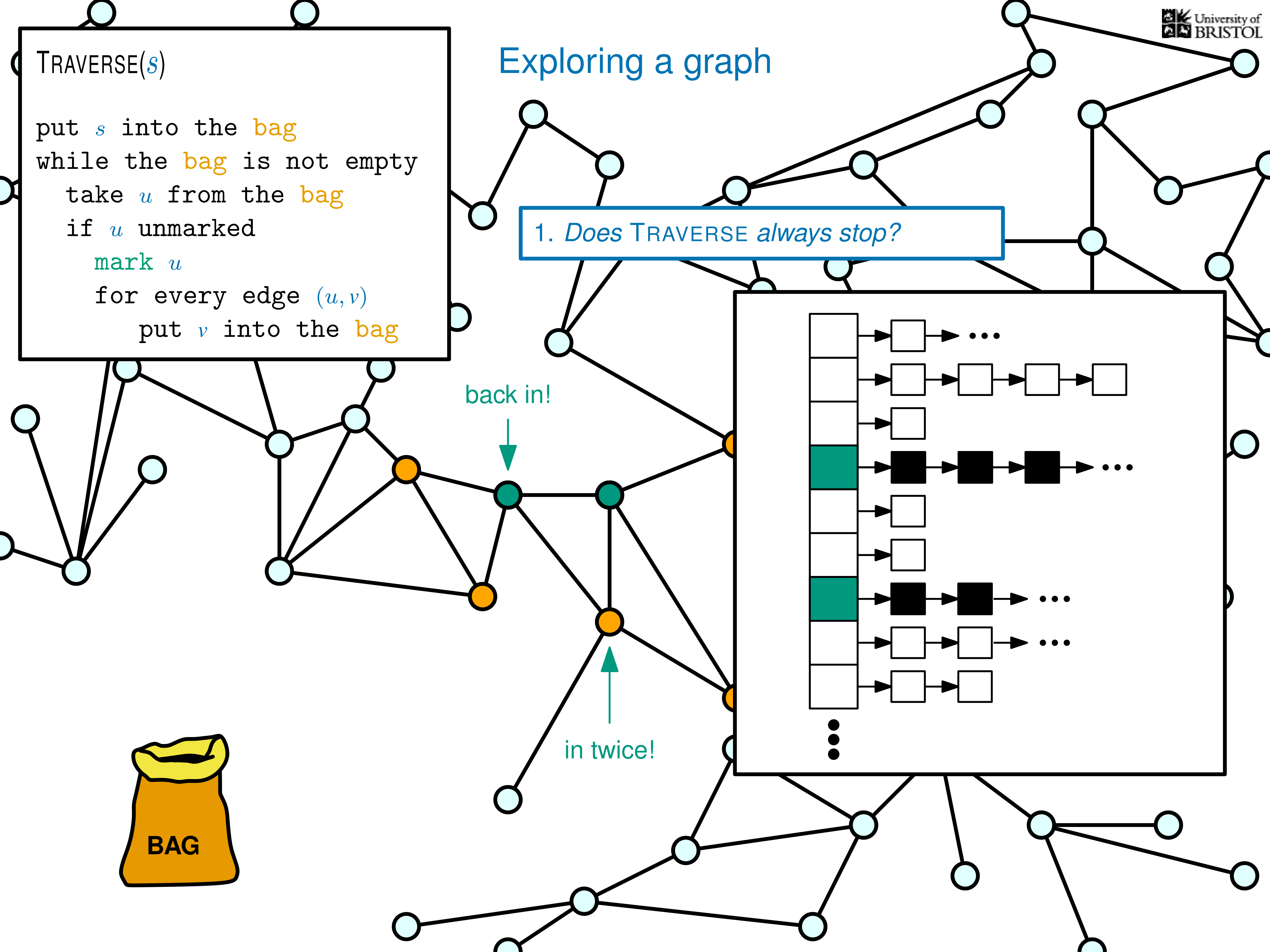


Exploring a graph

TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag

1. *Does* TRAVERSE *always stop?*

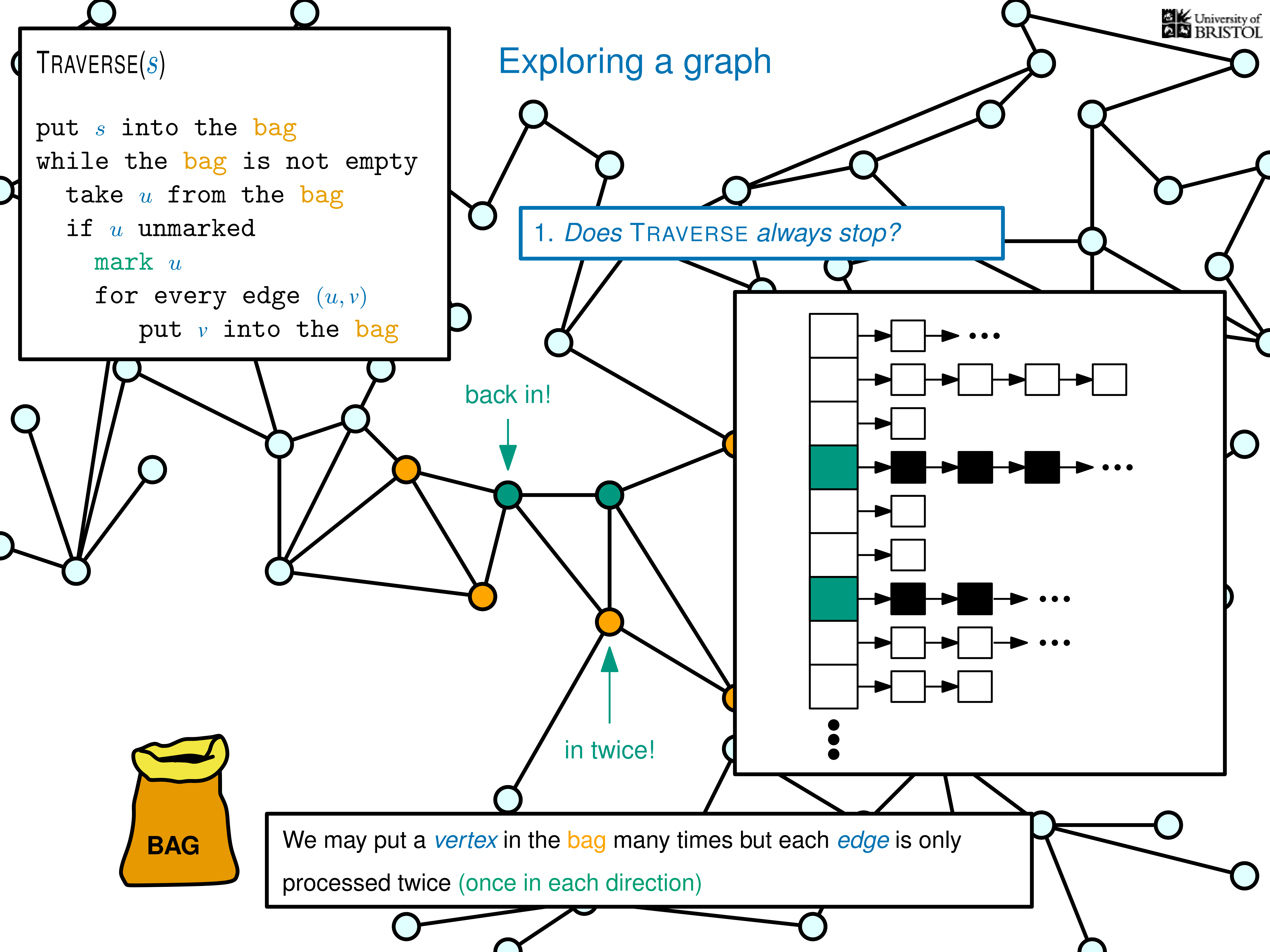


Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

1. *Does* TRAVERSE *always stop?*



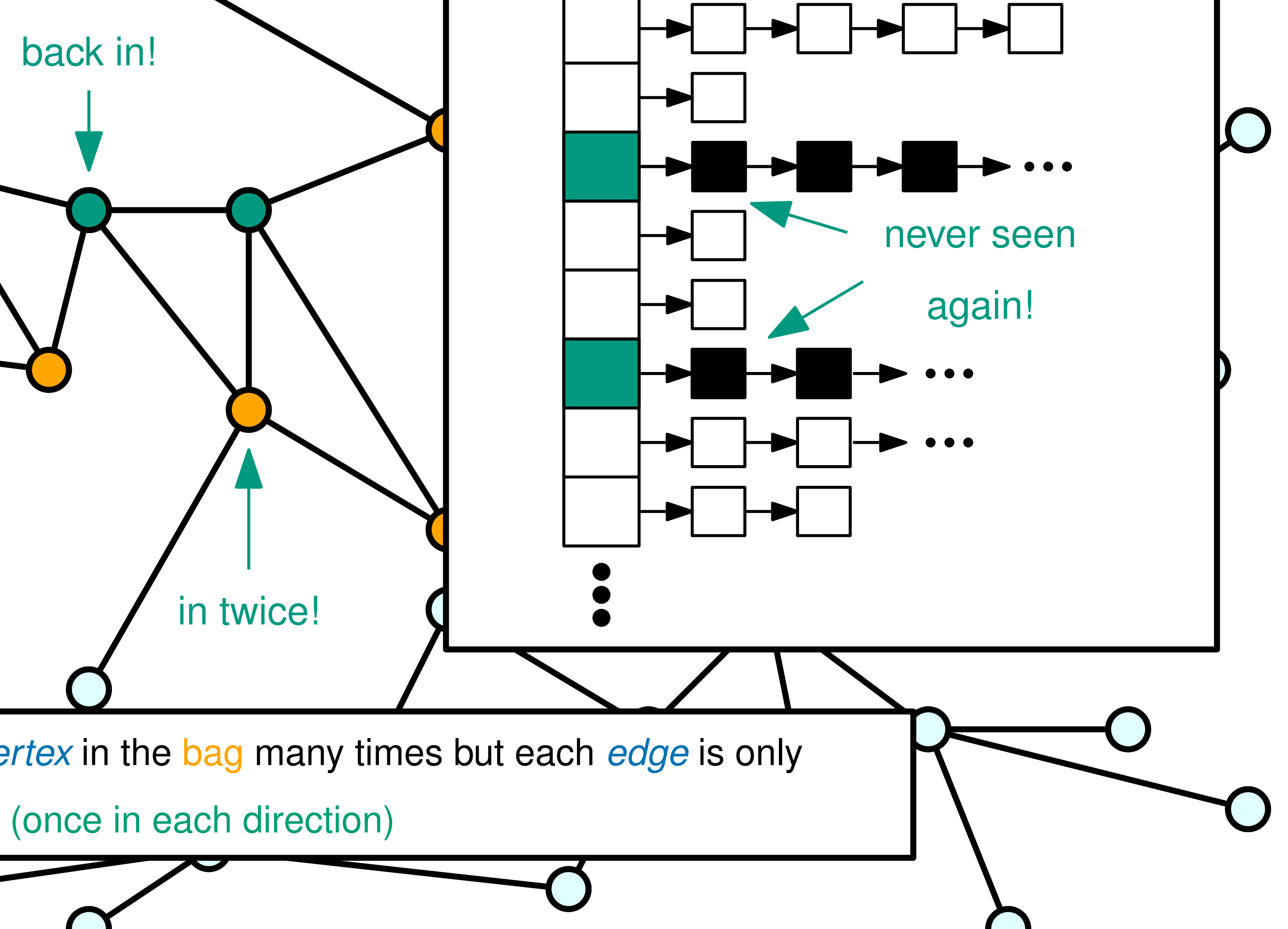
We may put a *vertex* in the bag many times but each *edge* is only processed twice (once in each direction)

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

1. *Does* TRAVERSE *always stop?*



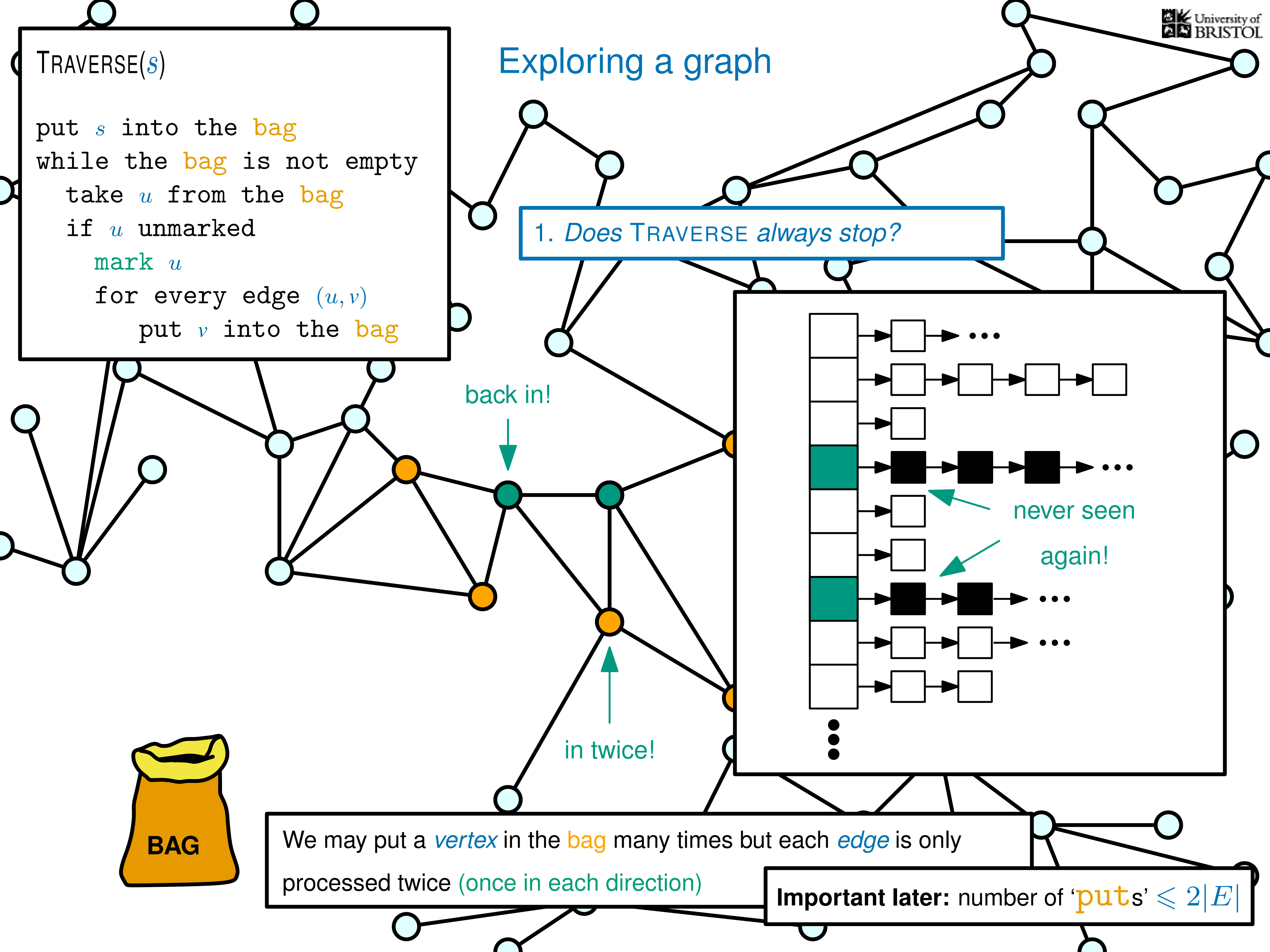
We may put a *vertex* in the bag many times but each *edge* is only processed twice (once in each direction)

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

1. *Does* TRAVERSE *always stop?*



We may put a *vertex* in the bag many times but each *edge* is only processed twice (once in each direction)

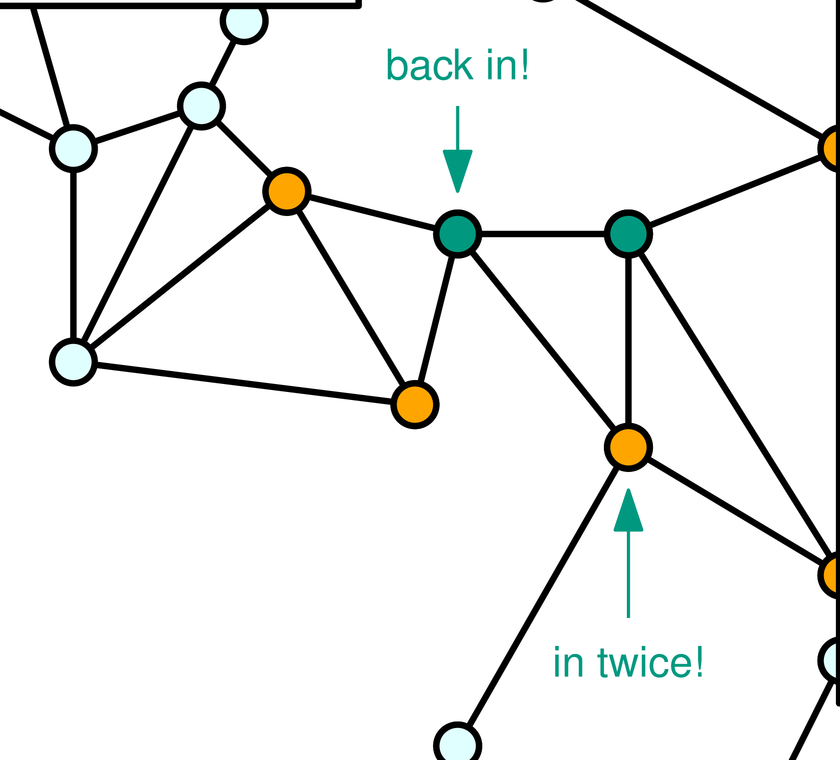
Important later: number of '`put`s' $\leqslant 2|E|$

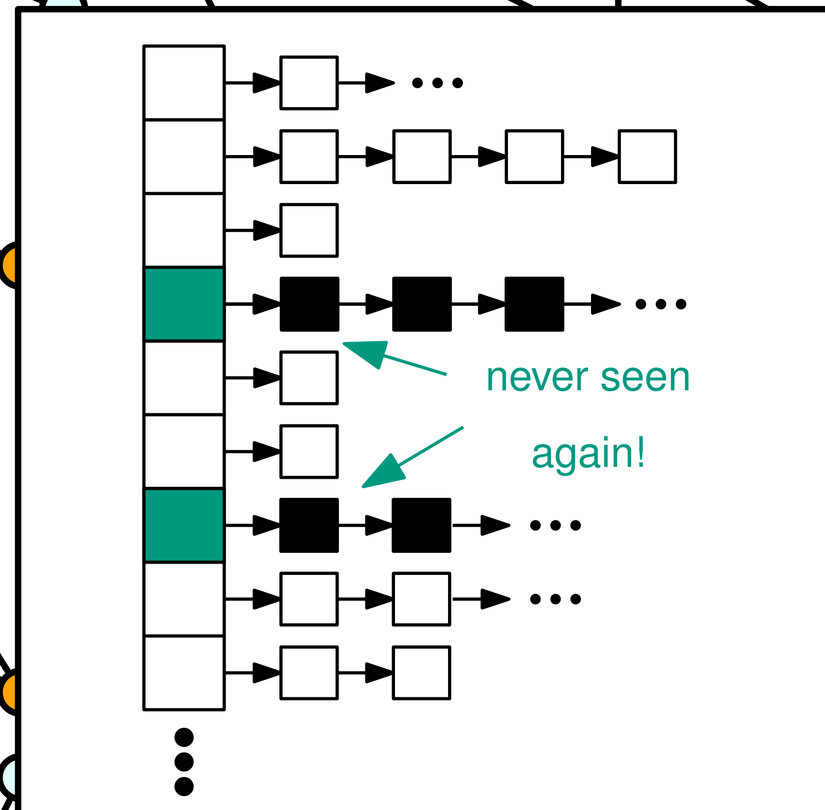
Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

1. *Does* TRAVERSE *always stop?*





We may put a *vertex* in the bag many times but each *edge* is only processed twice (once in each direction)

Important later: number of ‘puts’ $\leqslant 2|E|$

Exploring a graph

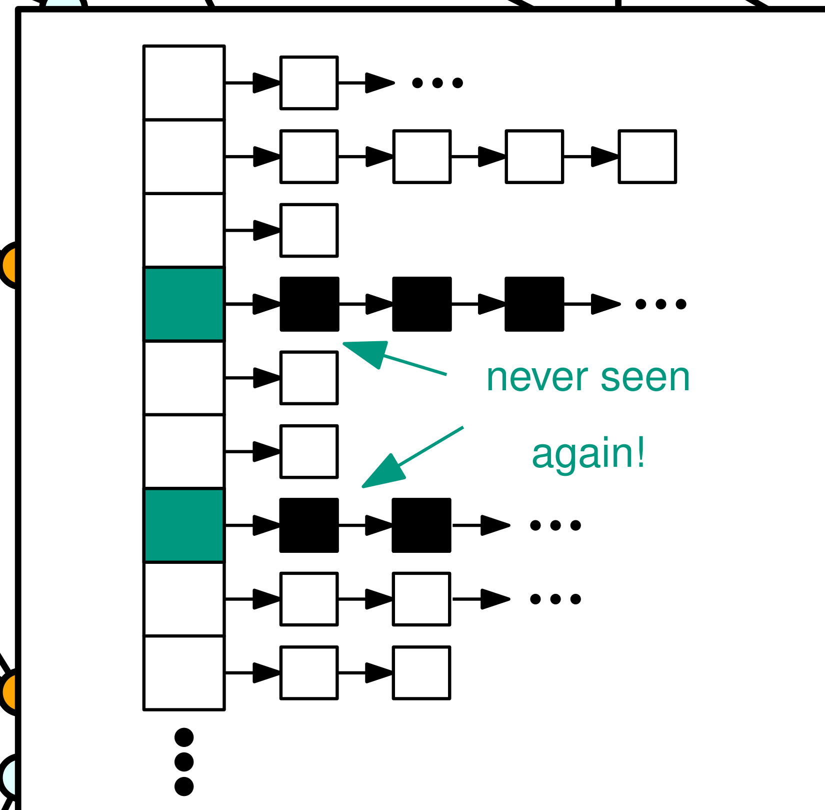
TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

1. *Does* TRAVERSE *always stop?*

back in!

in twice!



We may put a *vertex* in the bag many times but each *edge* is only processed twice (once in each direction)

Important later: number of 'puts' $\leqslant 2|E|$

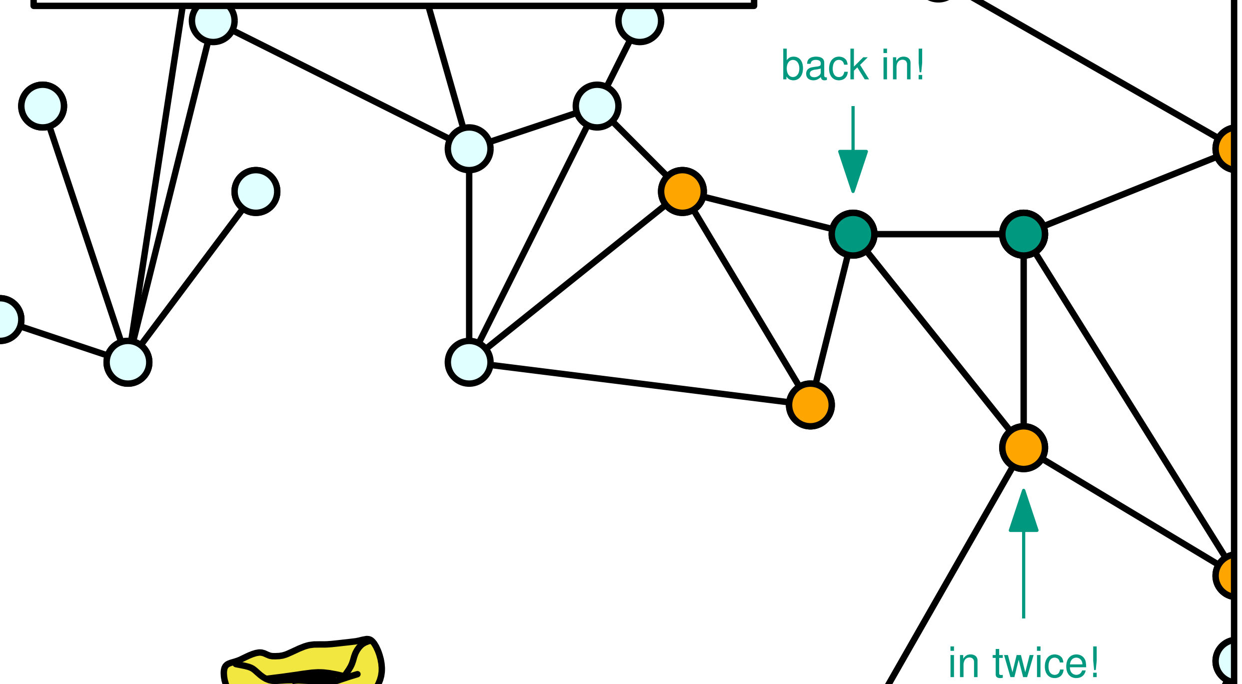
Exploring a graph

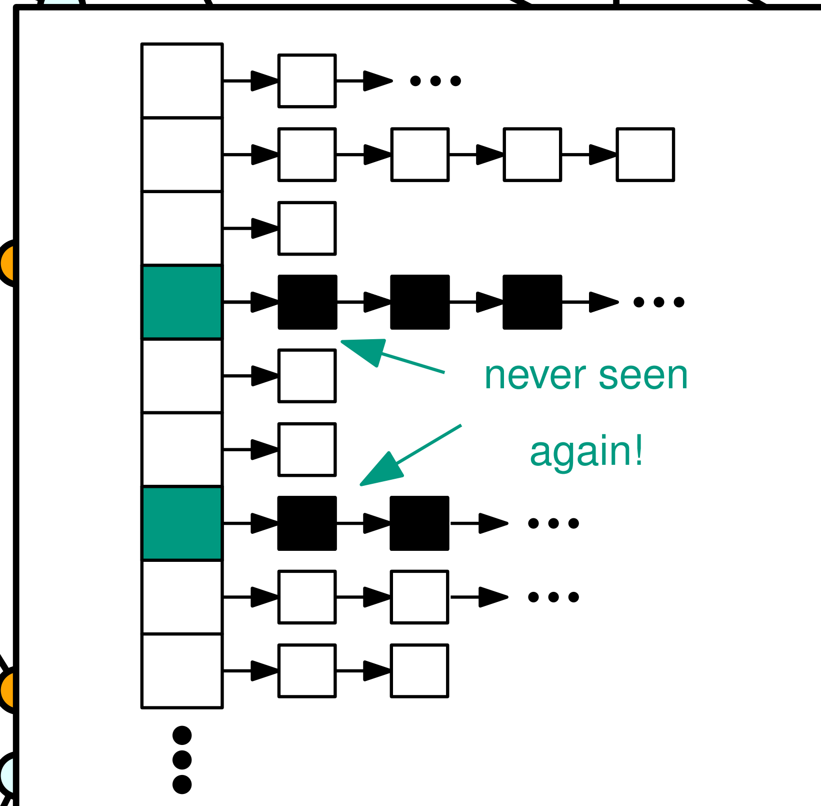
```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

1. *Does* TRAVERSE *always stop?*

Yes

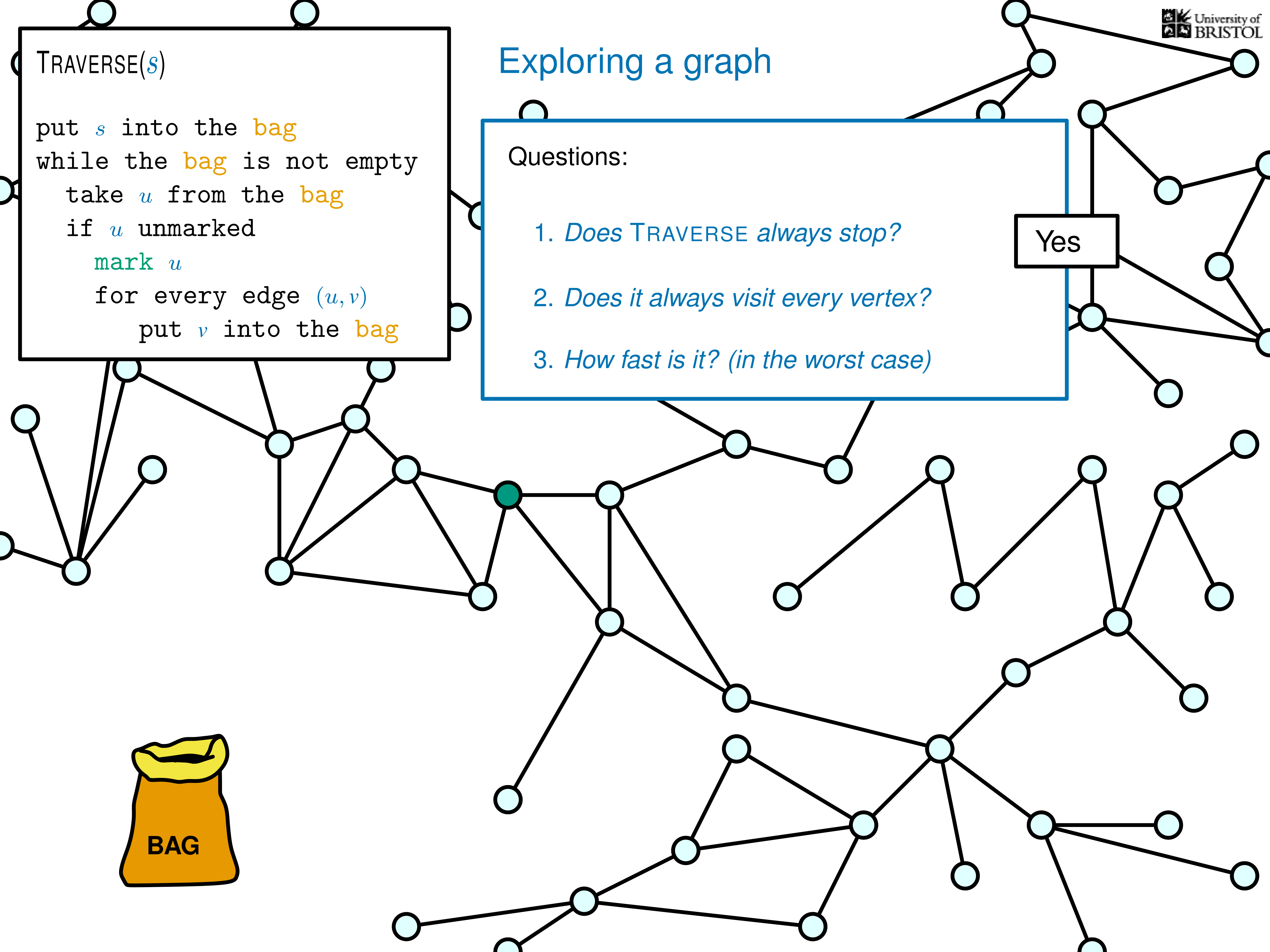




We may put a *vertex* in the bag many times but each *edge* is only processed twice (once in each direction)

Important later: number of 'puts' $\leqslant 2|E|$

Exploring a graph



TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

Questions:

1. *Does* TRAVERSE *always stop?*

Yes

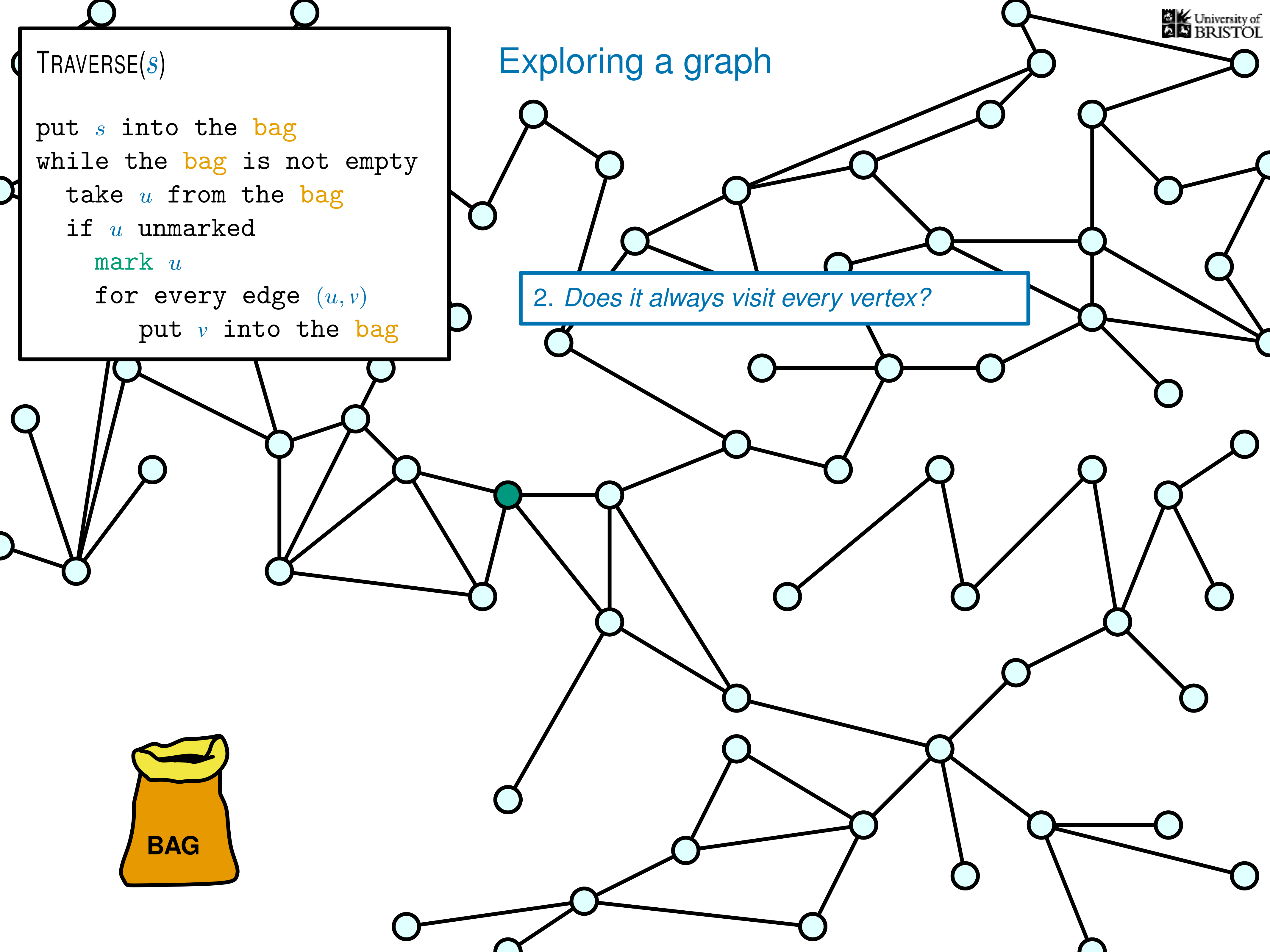
2. *Does it always visit every vertex?*
3. *How fast is it? (in the worst case)*

Exploring a graph

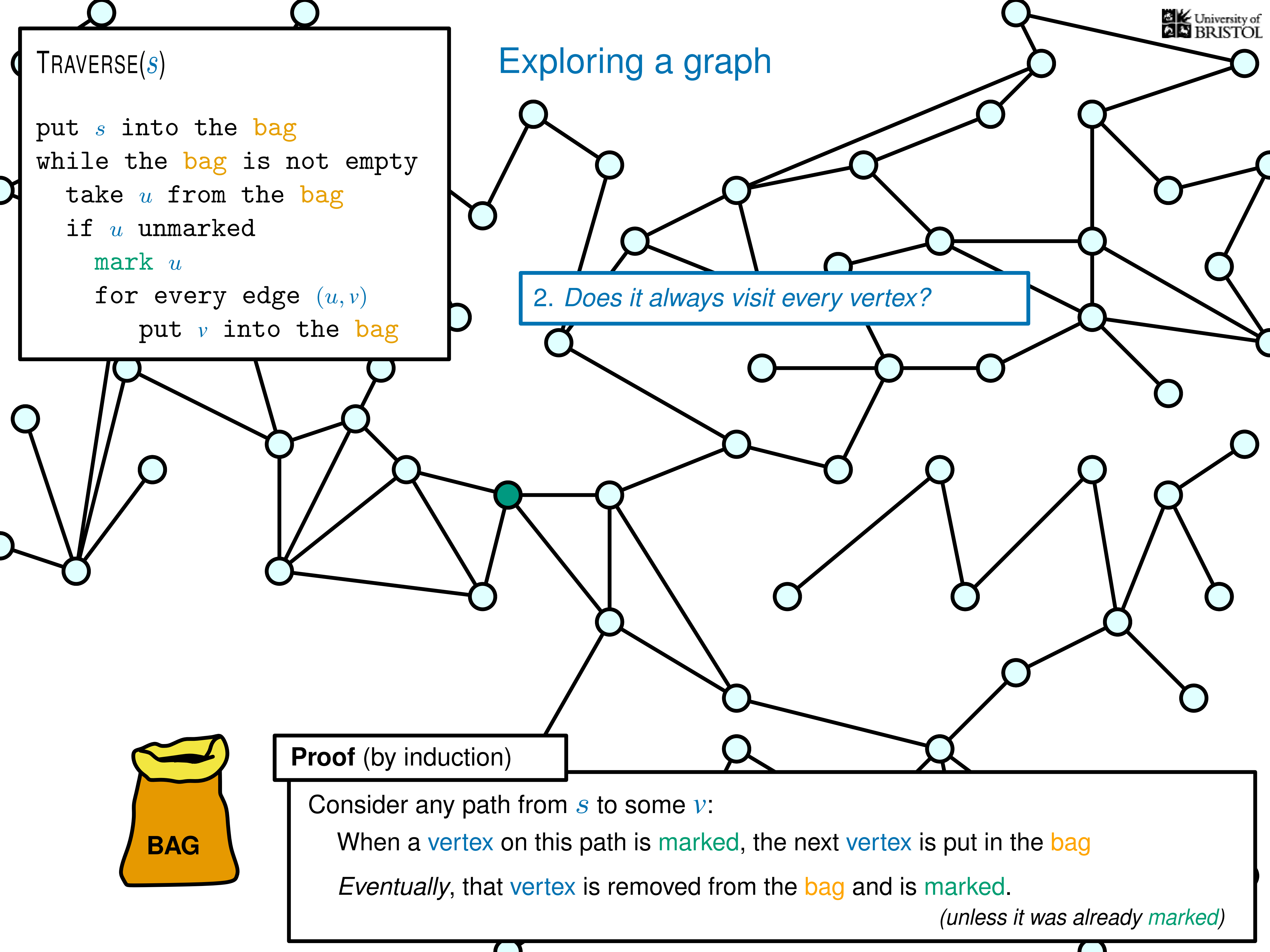
TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Exploring a graph



TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

BAG

2. *Does it always visit every vertex?*

Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

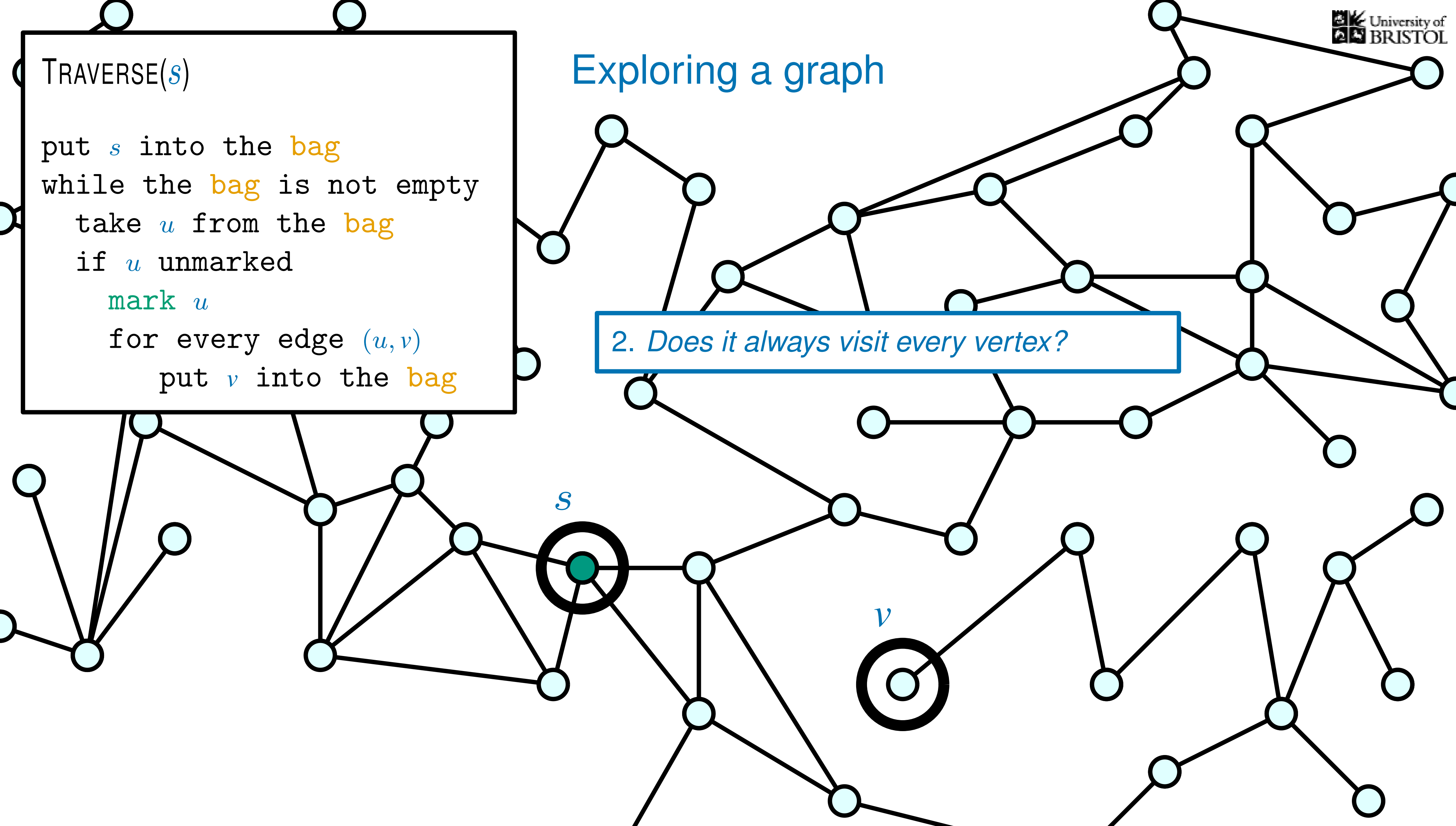
(unless it was already marked)

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

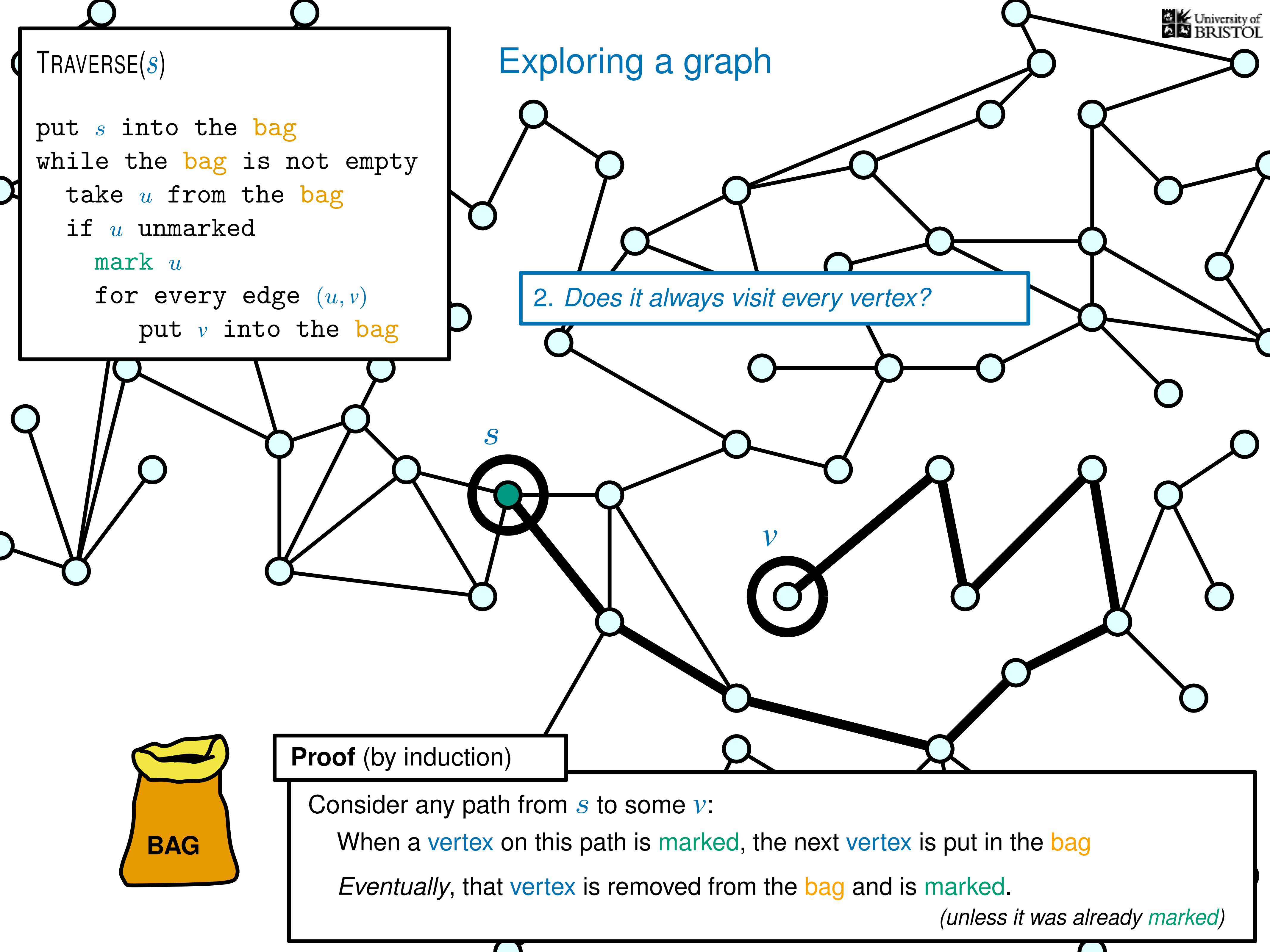
(unless it was already marked)

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

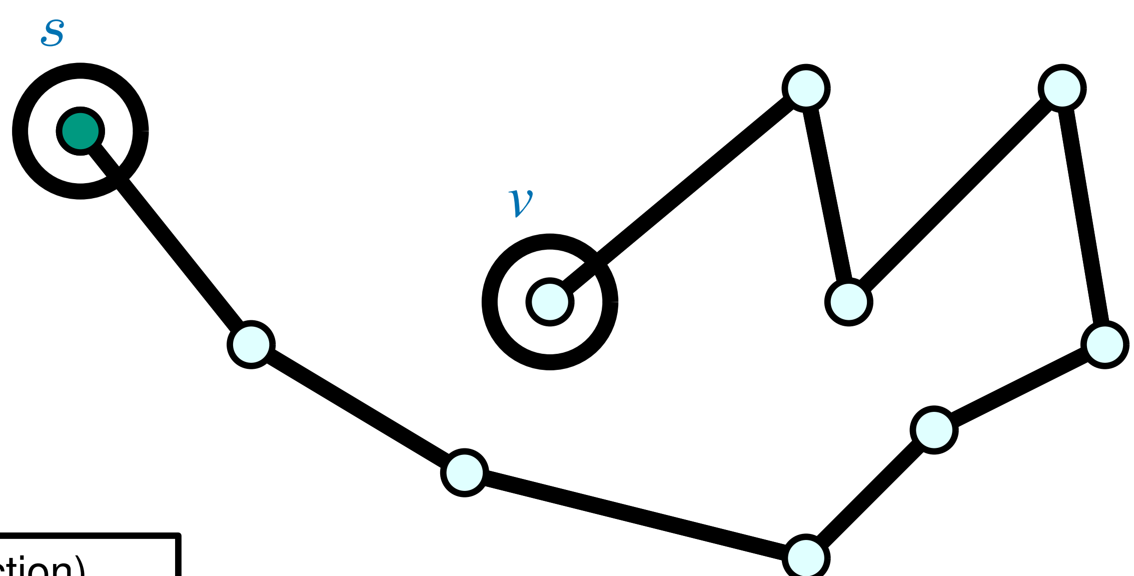
(unless it was already marked)

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

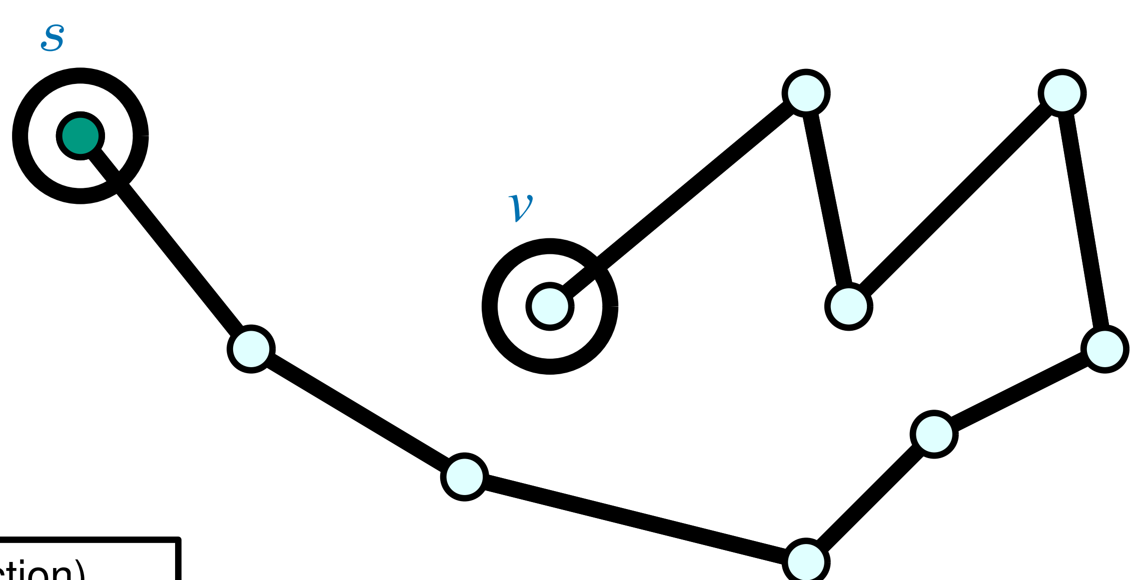
(unless it was already marked)

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

Exploring a graph

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

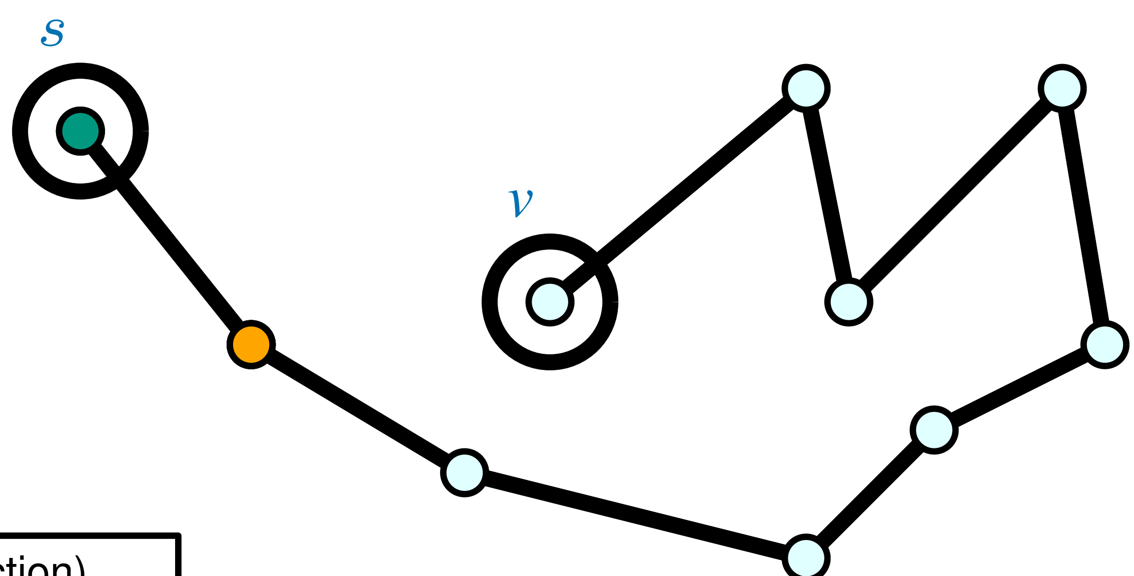
(unless it was already marked)

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

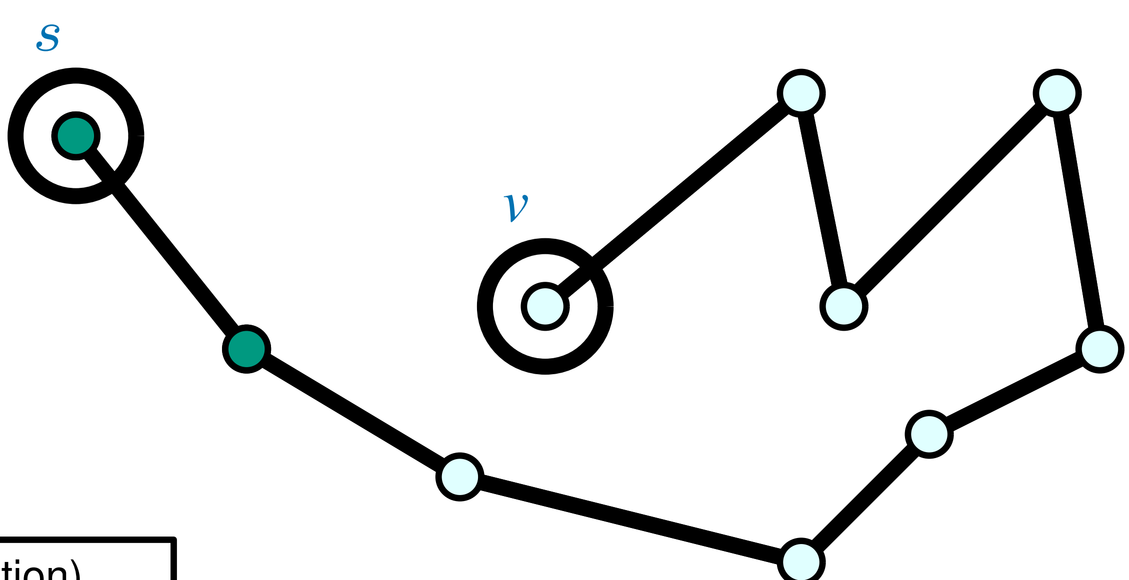
(unless it was already marked)

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

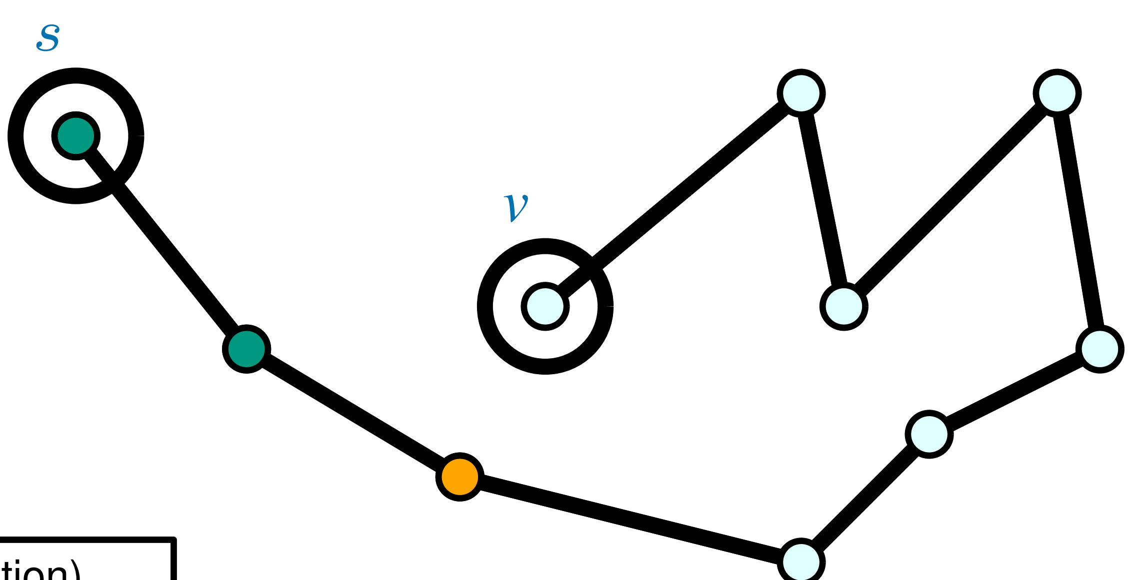
(unless it was already marked)

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
      put v into the bag
```

Exploring a graph

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

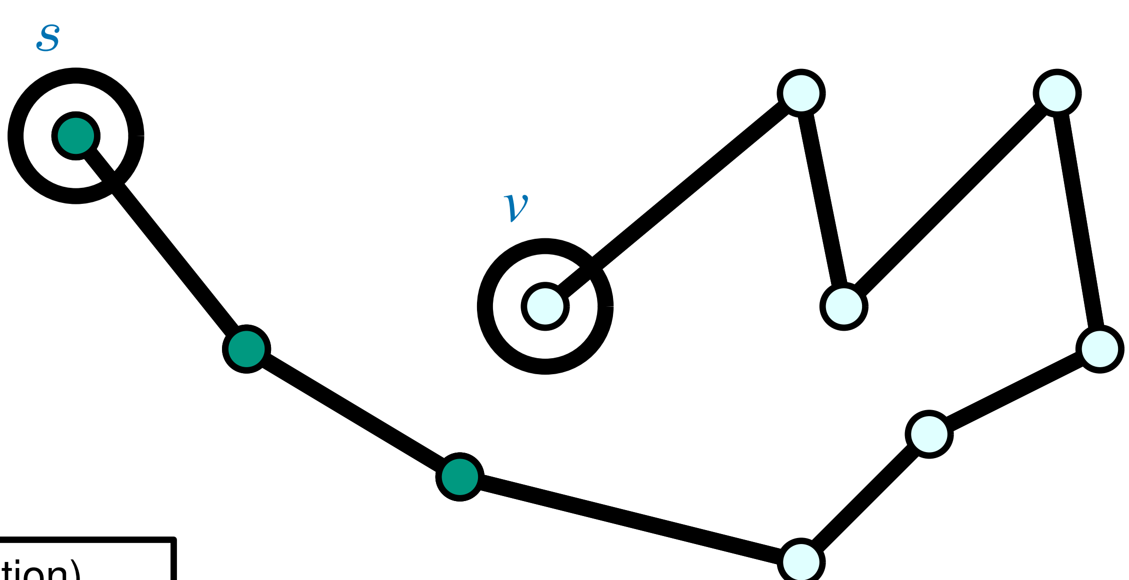
(unless it was already marked)

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

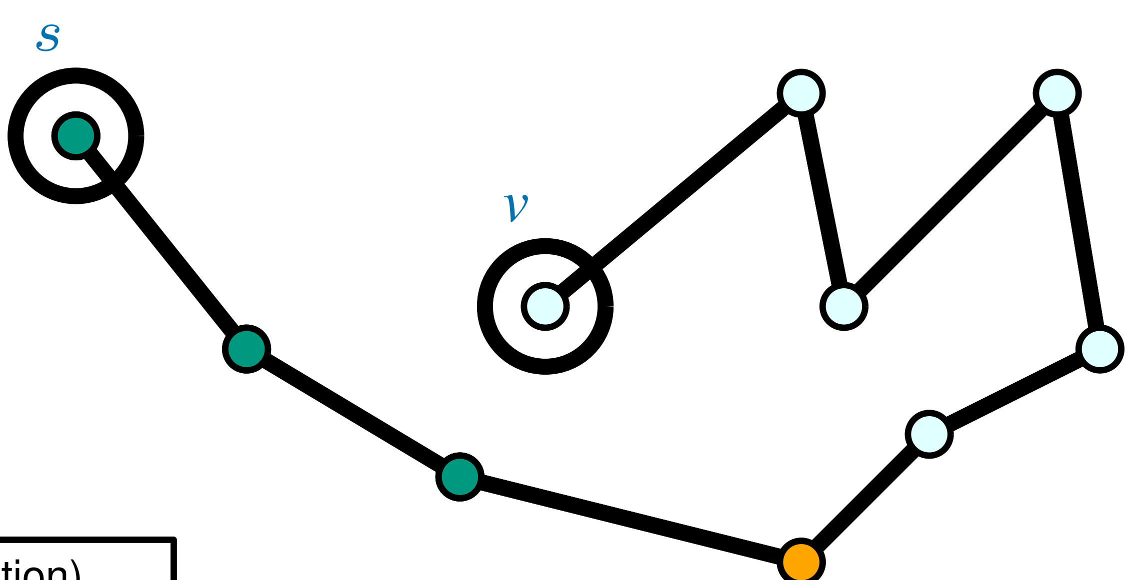
(unless it was already marked)

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

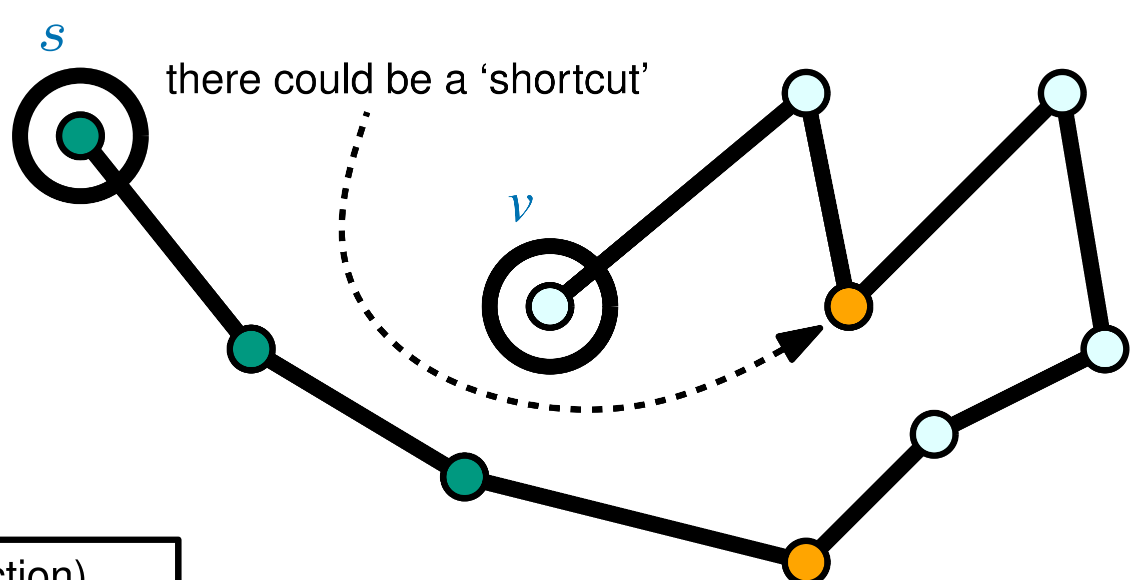
(unless it was already marked)

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

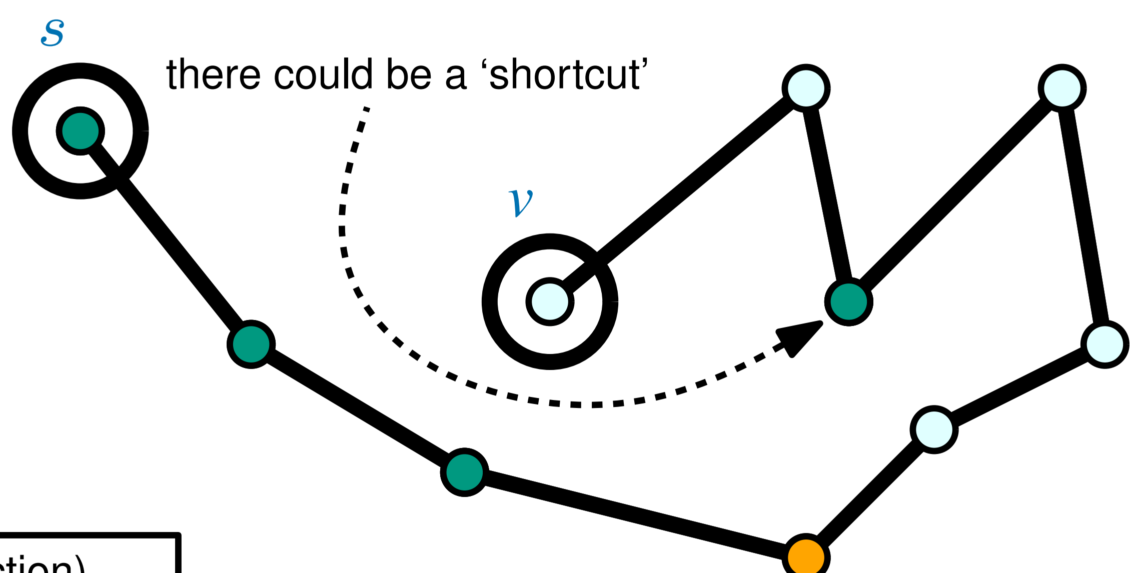
(unless it was already marked)

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

Exploring a graph

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

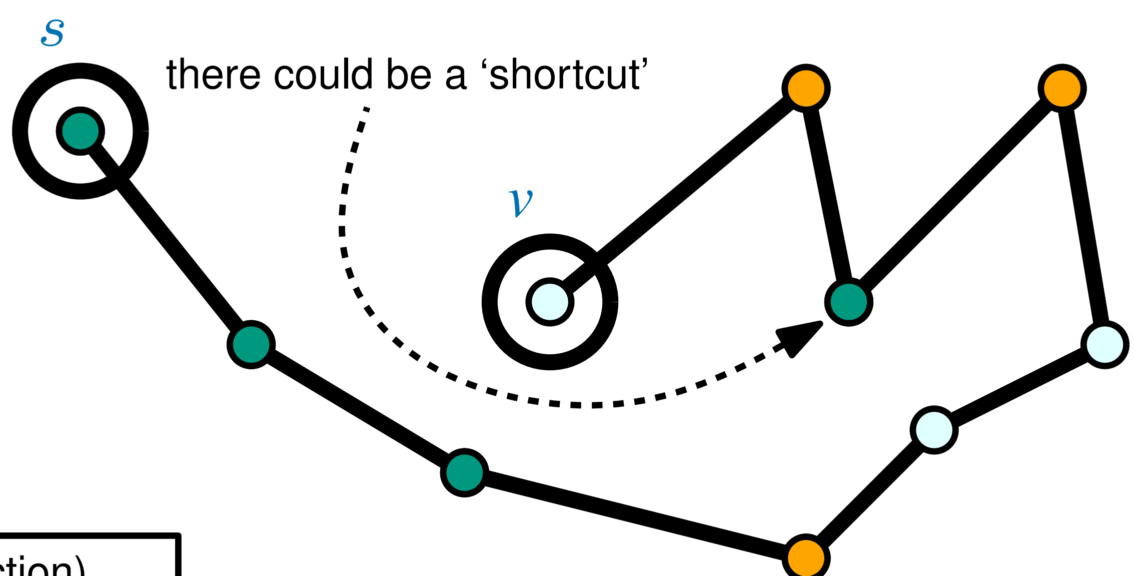
(unless it was already marked)

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

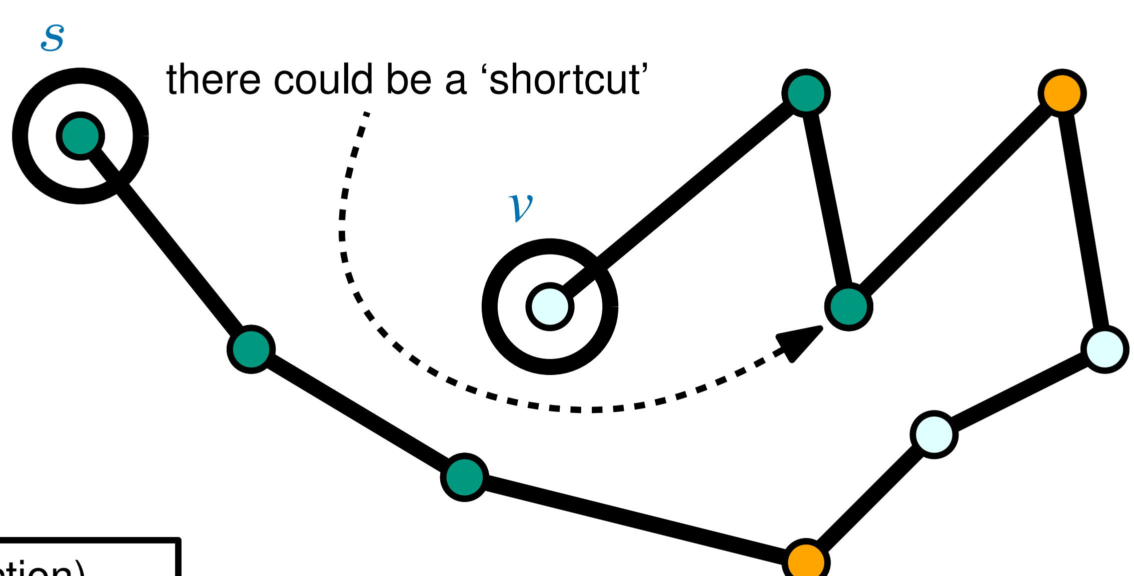
(unless it was already marked)

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

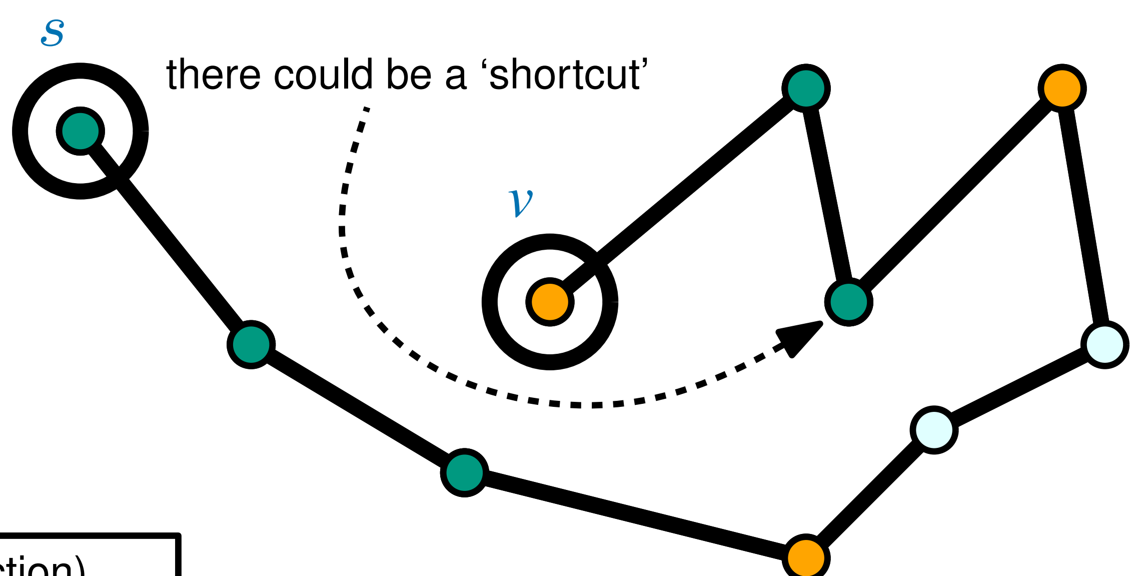
(unless it was already marked)

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

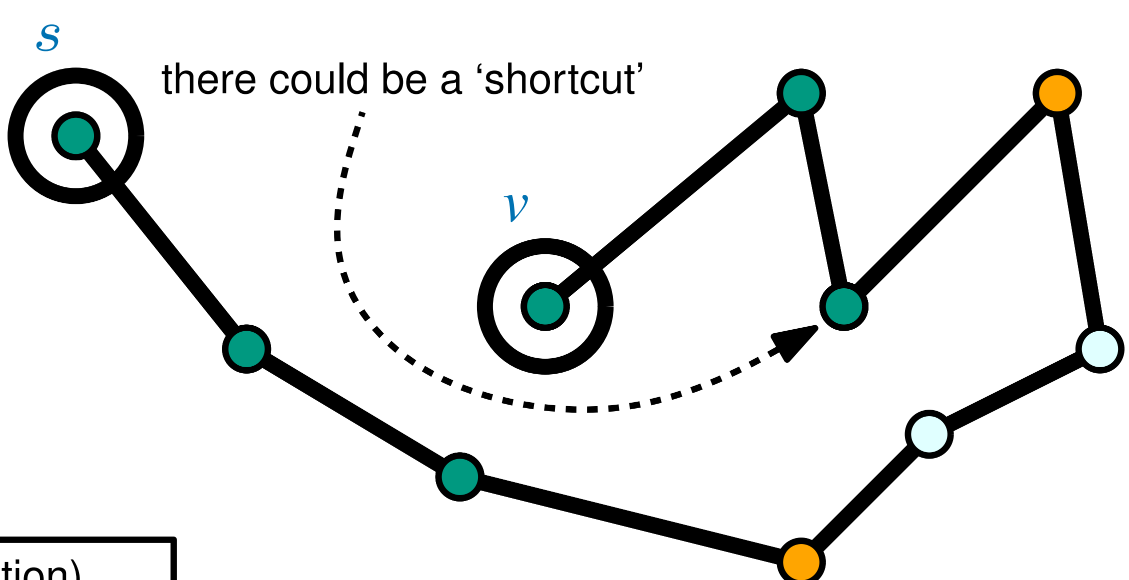
(unless it was already marked)

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

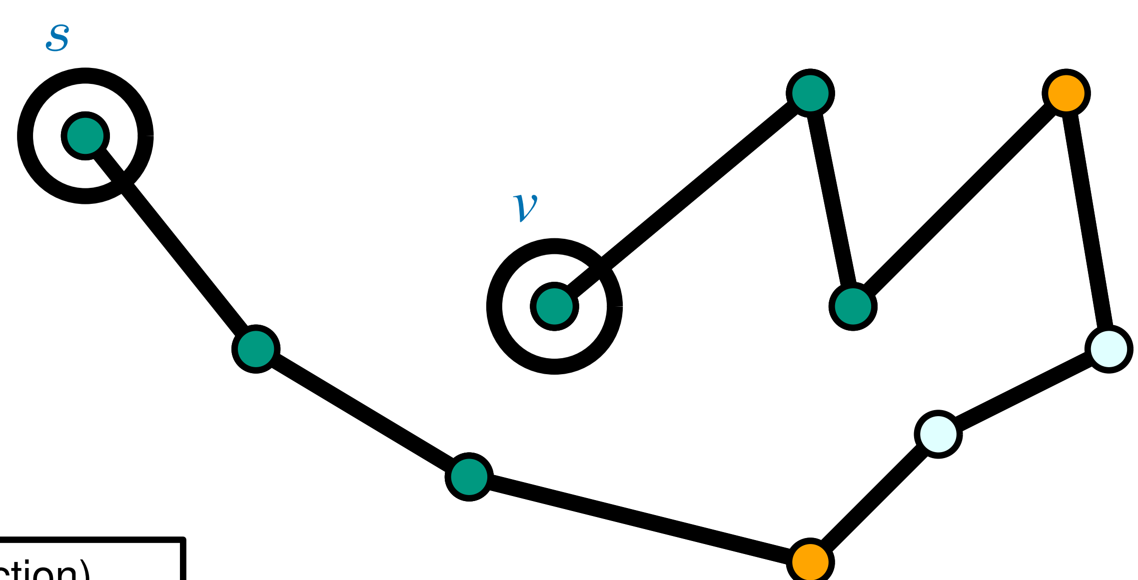
(unless it was already marked)

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

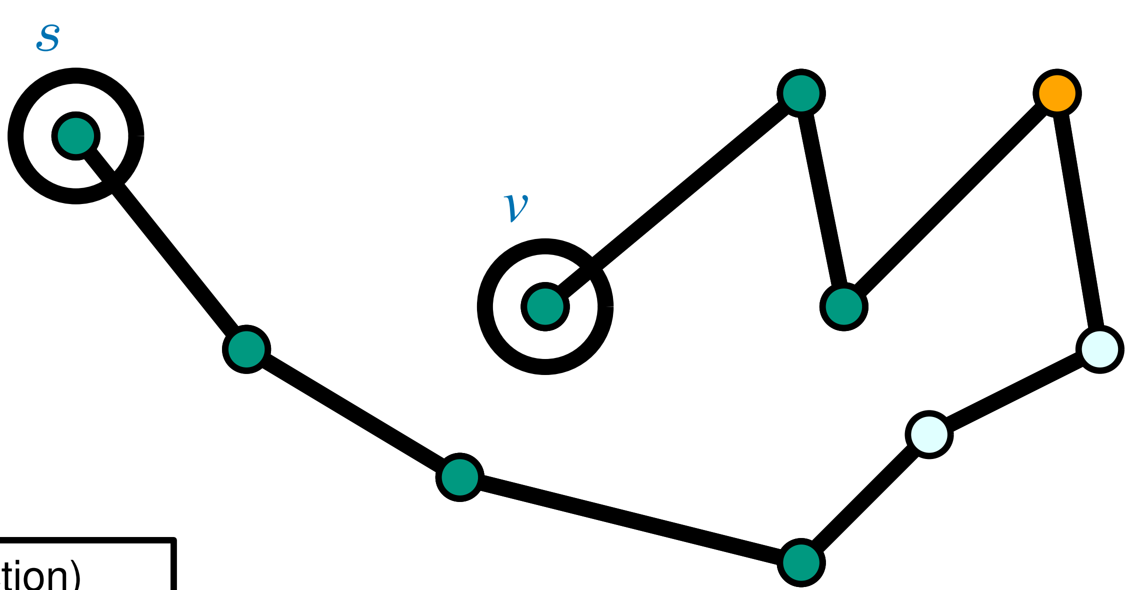
(unless it was already marked)

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

Exploring a graph

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

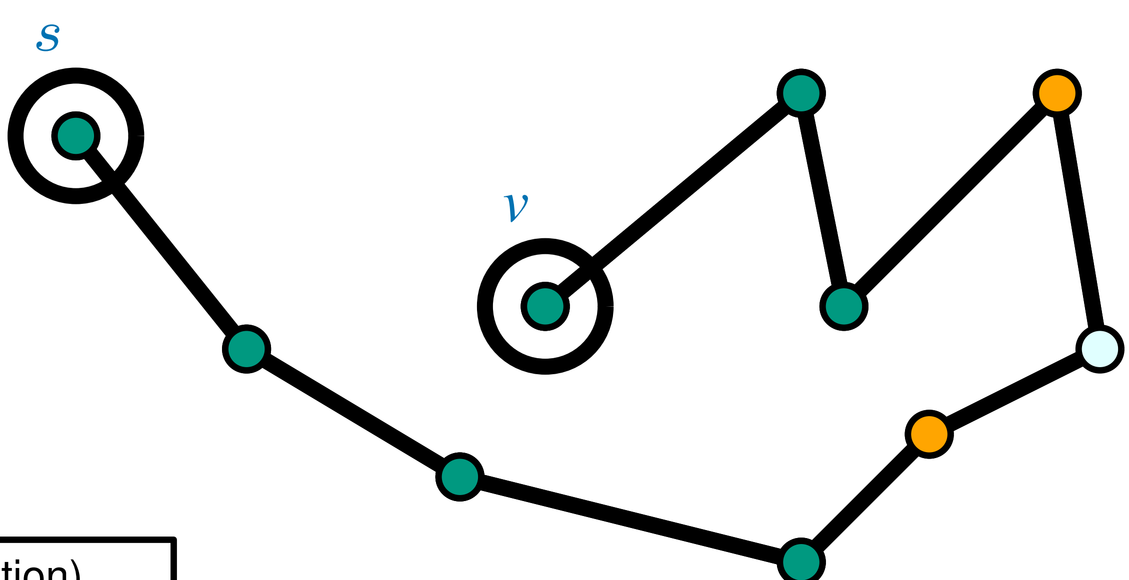
(unless it was already marked)

Exploring a graph

$\textsc{Traverse}(s)$

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

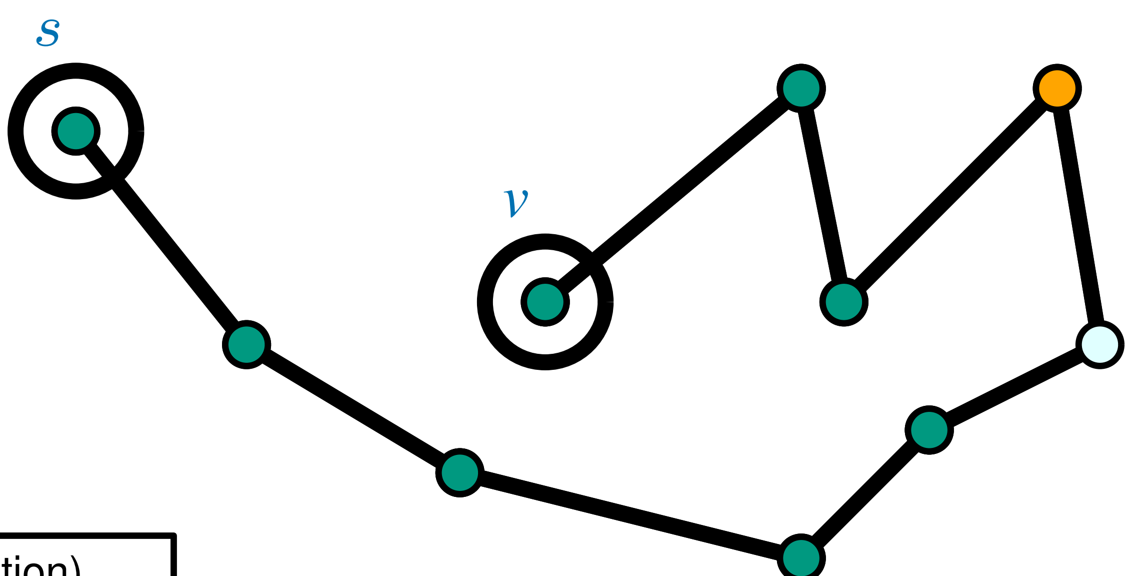
(unless it was already marked)

Exploring a graph

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

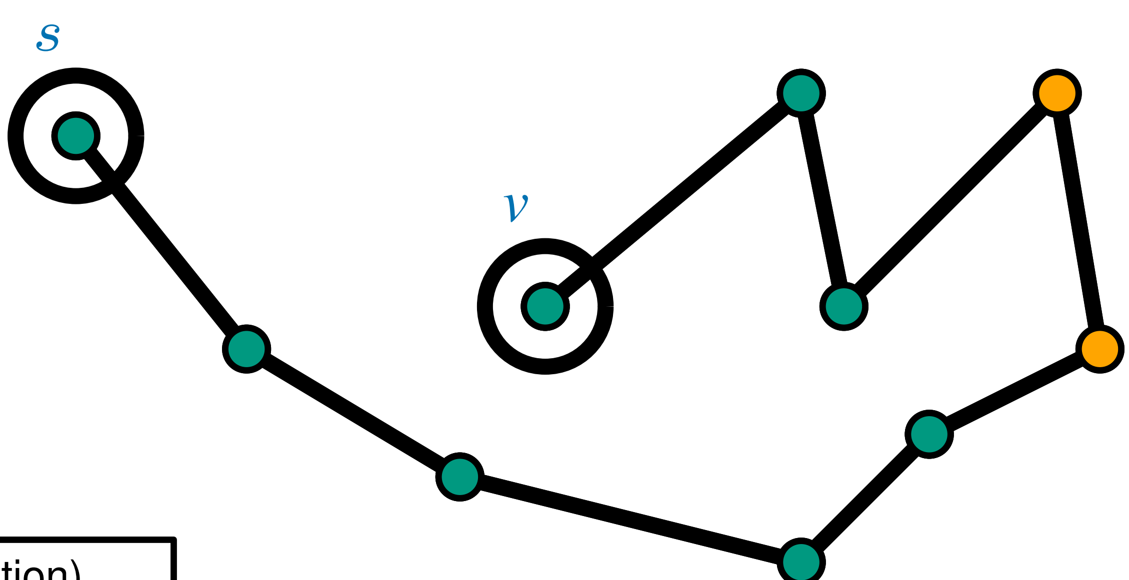
(unless it was already marked)

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

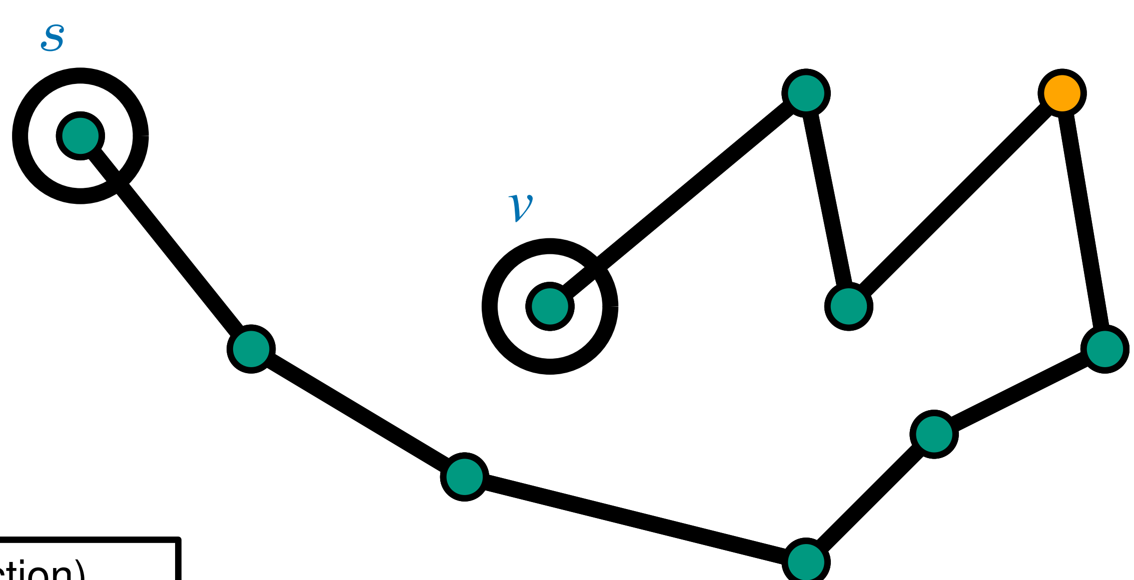
(unless it was already marked)

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

Exploring a graph

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

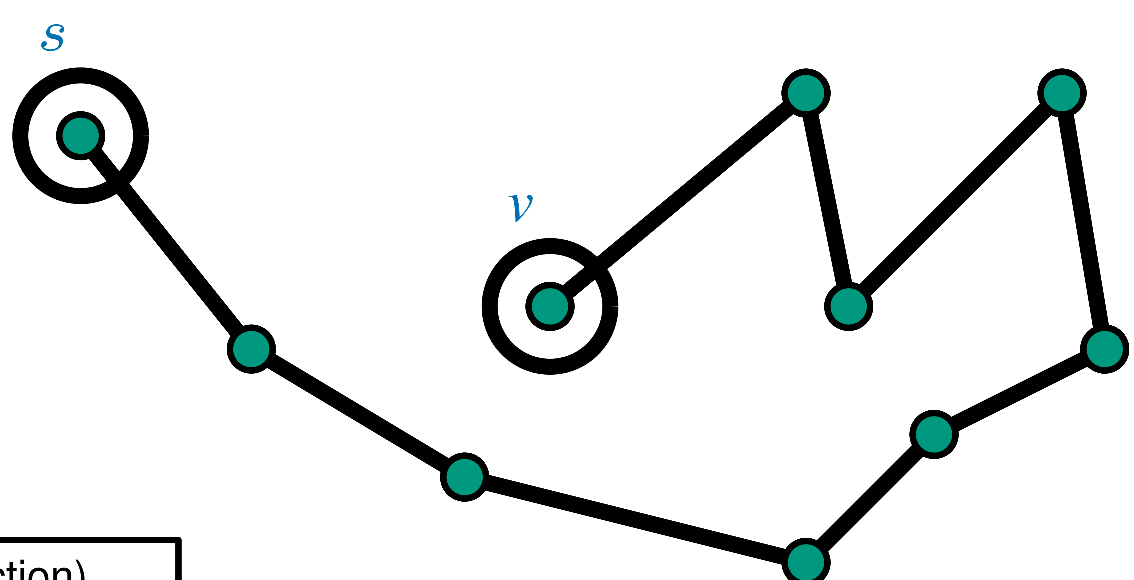
(unless it was already marked)

Exploring a graph

```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

(unless it was already marked)

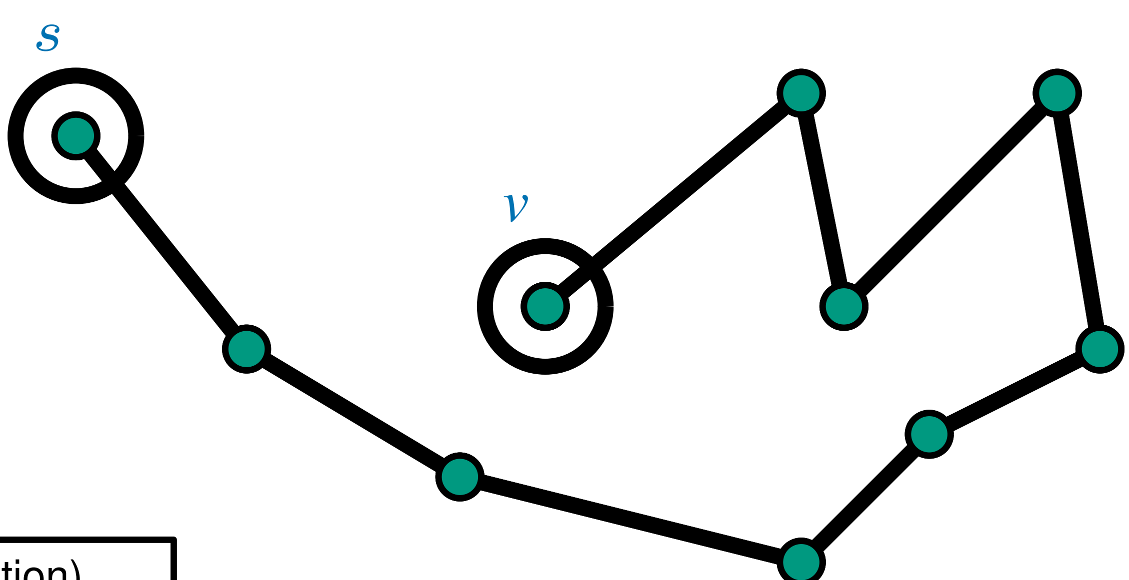
```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

Exploring a graph

2. *Does it always visit every vertex?*

Yes



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

(unless it was already marked)

Exploring a graph

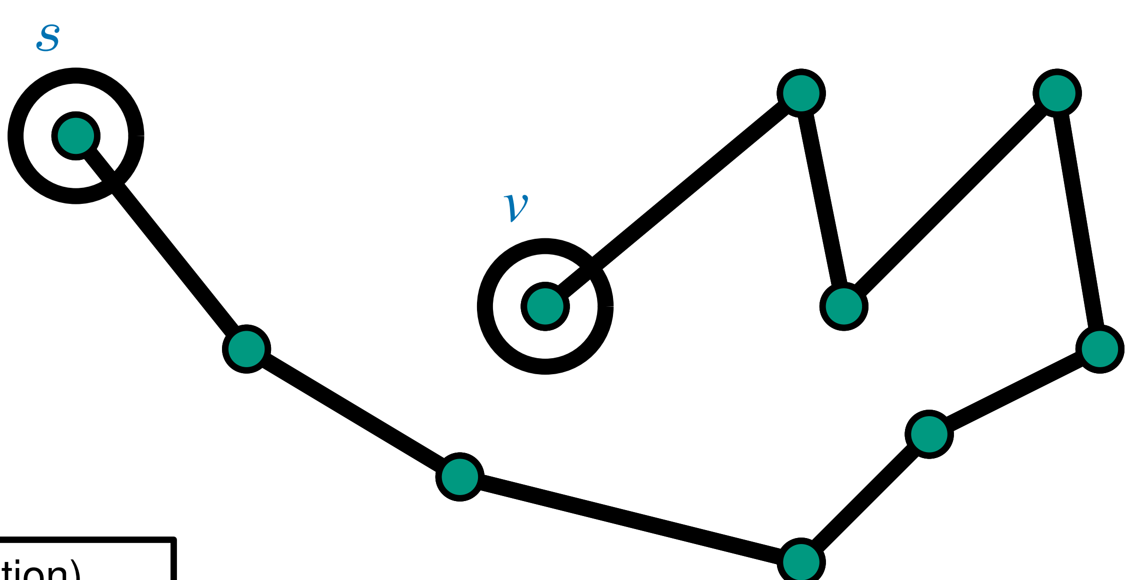
```
TRAVERSE(s)

put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

2. *Does it always visit every vertex?*

Yes

The correctness doesn't depend on how the bag works!



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

(unless it was already marked)

Exploring a graph

TRAVERSE(s)

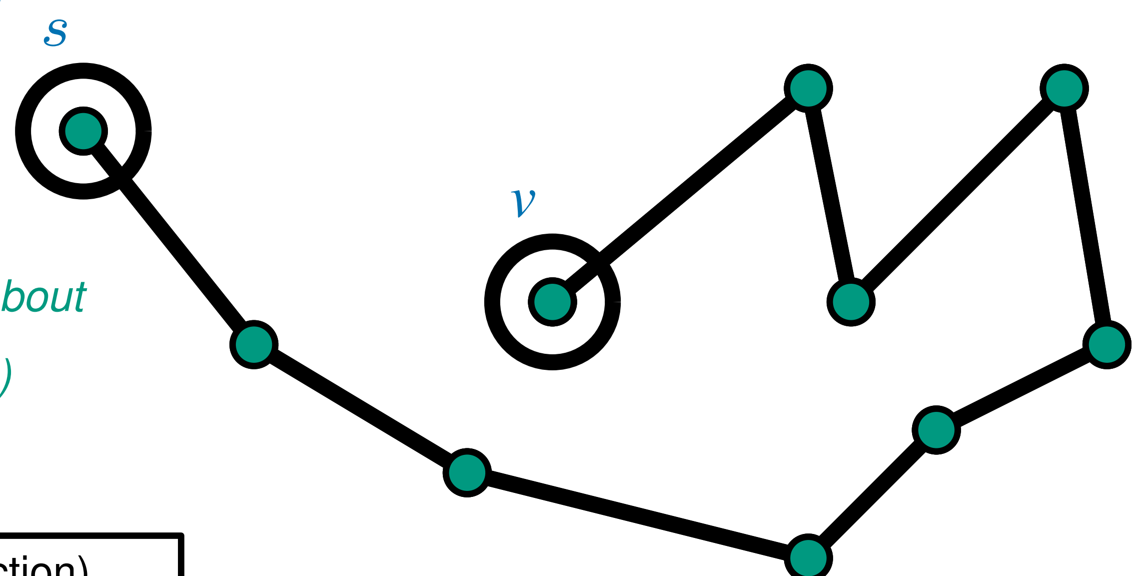
```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

2\. *Does it always visit every vertex?*

Yes

The correctness doesn't depend on how the bag works!

(but this doesn't tell you anything about which order vertices are marked in)



Proof (by induction)

Consider any path from s to some v:

When a vertex on this path is marked, the next vertex is put in the bag

Eventually, that vertex is removed from the bag and is marked.

(unless it was already marked)

Exploring a graph

Questions:

1. *Does* TRAVERSE *always stop?*
2. *Does it always visit every vertex?*
3. *How fast is it? (in the worst case)*

Yes

Yes

Exploring a graph

Assumption

bag operations take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* $\boxed{\text{Yes}}$
2. *Does it always visit every vertex?* $\boxed{\text{Yes}}$
3. *How fast is it? (in the worst case)*

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Questions:

1. *Does* TRAVERSE *always stop?*
2. *Does it always visit every vertex?*
3. *How fast is it? (in the worst case)*

Yes

Yes

Assumption

bag operations take $O(1)$ time

(we'll come back to this)

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* — Yes
2. *Does it always visit every vertex?* — Yes
3. *How fast is it? (in the worst case)*

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the bag
```

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Assumption

bag operations take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* Yes
2. *Does it always visit every vertex?* Yes
3. *How fast is it? (in the worst case)*

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

Time complexity:

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* Yes
2. *Does it always visit every vertex?* Yes
3. *How fast is it? (in the worst case)*

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

Time complexity:

$O(1)$ time every time we `take` from the bag

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Assumption

bag operations take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* Yes
2. *Does it always visit every vertex?* Yes
3. *How fast is it? (in the worst case)*

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
   take u from the bag
   if u unmarked
      mark u
      for every edge (u,v)
         put v into the bag
```

Time complexity:

$O(1)$ time every time we `take` from the bag

(store an array where $\texttt{mark}[u] = 1$ *iff* u *is marked)*

Exploring a graph

V is the set of vertices
$|V|$ *is the number of vertices*

E is the set of edges
$|E|$ *is the number of edges*

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* — Yes
2. *Does it always visit every vertex?* — Yes
3. *How fast is it? (in the worst case)*

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
   take u from the bag
   if u unmarked
      mark u
      for every edge (u, v)
         put v into the bag
```

Time complexity:

$O(1)$ time every time we `take` from the bag

(store an array where `mark[`u`]` $= 1$ *iff* u *is marked)*

$O(1)$ time every time we `put` into the bag

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Questions:

1. *Does* TRAVERSE *always stop?* Yes
2. *Does it always visit every vertex?* Yes
3. *How fast is it? (in the worst case)*

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
   take u from the bag
   if u unmarked
      mark u
      for every edge (u, v)
         put v into the bag
```

Time complexity:

$O(1)$ time every time we `take` from the bag

(store an array where `mark`$[u] = 1$ *iff* u *is marked)*

$O(1)$ time every time we `put` into the bag

What is the total number of bag operations?

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Questions:

1. *Does* TRAVERSE *always stop?* — Yes
2. *Does it always visit every vertex?* — Yes
3. *How fast is it? (in the worst case)*

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
   take u from the bag
   if u unmarked
      mark u
      for every edge (u, v)
         put v into the bag
```

Time complexity:

$O(1)$ time every time we `take` from the bag

(store an array where `mark`$[u] = 1$ *iff* u *is marked)*

$O(1)$ time every time we `put` into the bag

What is the total number of bag operations?

we do at most $2|E|$ `put` operations

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Questions:

1. *Does* TRAVERSE *always stop?* **Yes**
2. *Does it always visit every vertex?* **Yes**
3. *How fast is it? (in the worst case)*

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
   take u from the bag
   if u unmarked
      mark u
      for every edge (u, v)
         put v into the bag
```

Time complexity:

$O(1)$ time every time we `take` from the bag

(store an array where `mark`$[u] = 1$ *iff* u *is marked)*

$O(1)$ time every time we `put` into the bag

What is the total number of bag operations?

we do at most $2|E|$ `put` operations

so we do at most $2|E|$ `take` operations

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Assumption

bag operations take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* Yes
2. *Does it always visit every vertex?* Yes
3. *How fast is it? (in the worst case)*

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

Time complexity:

$O(1)$ time every time we `take` from the bag

(store an array where `mark`$[u] = 1$ *iff* u *is marked)*

$O(1)$ time every time we `put` into the bag

What is the total number of bag operations?

we do at most $2|E|$ `put` operations

so we do at most $2|E|$ `take` operations

The overall time complexity is $O(|E|)$

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Questions:

1. *Does* TRAVERSE *always stop?* — Yes
2. *Does it always visit every vertex?* — Yes
3. *How fast is it? (in the worst case)* — $O(|E|)$

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
      put v into the bag
```

Time complexity:

$O(1)$ time every time we `take` from the bag

(store an array where `mark`$[u] = 1$ *iff u is marked)*

$O(1)$ time every time we `put` into the bag

What is the total number of bag operations?

we do at most $2|E|$ `put` operations

so we do at most $2|E|$ `take` operations

The overall time complexity is $O(|E|)$

Exploring a graph

V is the set of vertices
$|V|$ *is the number of vertices*

E is the set of edges
$|E|$ *is the number of edges*

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* — Yes
2. *Does it always visit every vertex?* — Yes
3. *How fast is it? (in the worst case)* — $O(|E|)$

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
        put v into the bag
```

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* — Yes
2. *Does it always visit every vertex?* — Yes
3. *How fast is it? (in the worst case)* — $O(|E|)$

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u,v)
      put v into the bag
```

What if we had used an adjacency matrix?

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

V is the set of vertices
$|V|$ *is the number of vertices*

Exploring a graph

E is the set of edges
$|E|$ *is the number of edges*

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* — Yes
2. *Does it always visit every vertex?* — Yes
3. *How fast is it? (in the worst case)* — $O(|E|)$

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

What if we had used an adjacency matrix?

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

finding all edges leaving u would have taken $O(|V|)$ time

Exploring a graph

V is the set of vertices
$|V|$ *is the number of vertices*

E is the set of edges
$|E|$ *is the number of edges*

University of Bristol

Assumption

bag operations
take $O(1)$ time

(we'll come back to this)

Questions:

1. *Does* TRAVERSE *always stop?* — Yes
2. *Does it always visit every vertex?* — Yes
3. *How fast is it? (in the worst case)* — $O(|E|)$

TRAVERSE(s)

```
put s into the bag
while the bag is not empty
  take u from the bag
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the bag
```

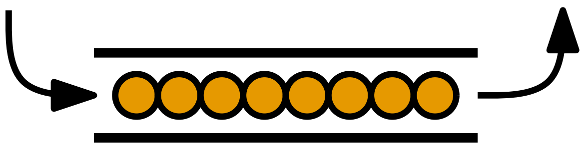
What if we had used an adjacency matrix?

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	1
3	1	1	0	1	1
4	0	1	1	0	1
5	0	1	1	1	0

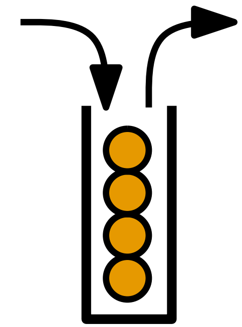
finding all edges leaving u would have taken $O(|V|)$ time

overall, the time complexity would have been $O(|V|^2)$ instead of $O(|E|)$.

How should we implement the bag?



Queue



Stack

Operations take $O(1)$ time

TRAVERSE visits every vertex in a connected graph in $O(|E|)$ time

but in different orders with different types of bag

with a Queue the algorithm is called

Breadth First Search

Applications

- **Shortest paths in unweighted graphs**
- Max-Flow
- Testing whether a graph is bipartite

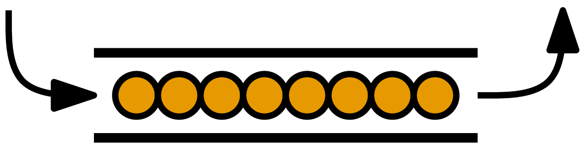
with a Stack the algorithm is called

Depth First Search

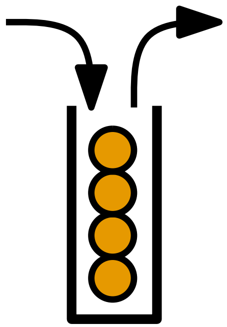
Applications

- Finding (strongly) connected components
- Topicologically sorting a Directed Acyclic Graph
- Testing for planarity

How should we implement the bag?



Queue



Stack

Operations take $O(1)$ time

TRAVERSE visits every vertex in a connected graph in $O(|E|)$ time

but in different orders with different types of bag

(and it works for directed graphs too)

with a Queue the algorithm is called

Breadth First Search

Applications

- **Shortest paths in unweighted graphs**
- Max-Flow
- Testing whether a graph is bipartite

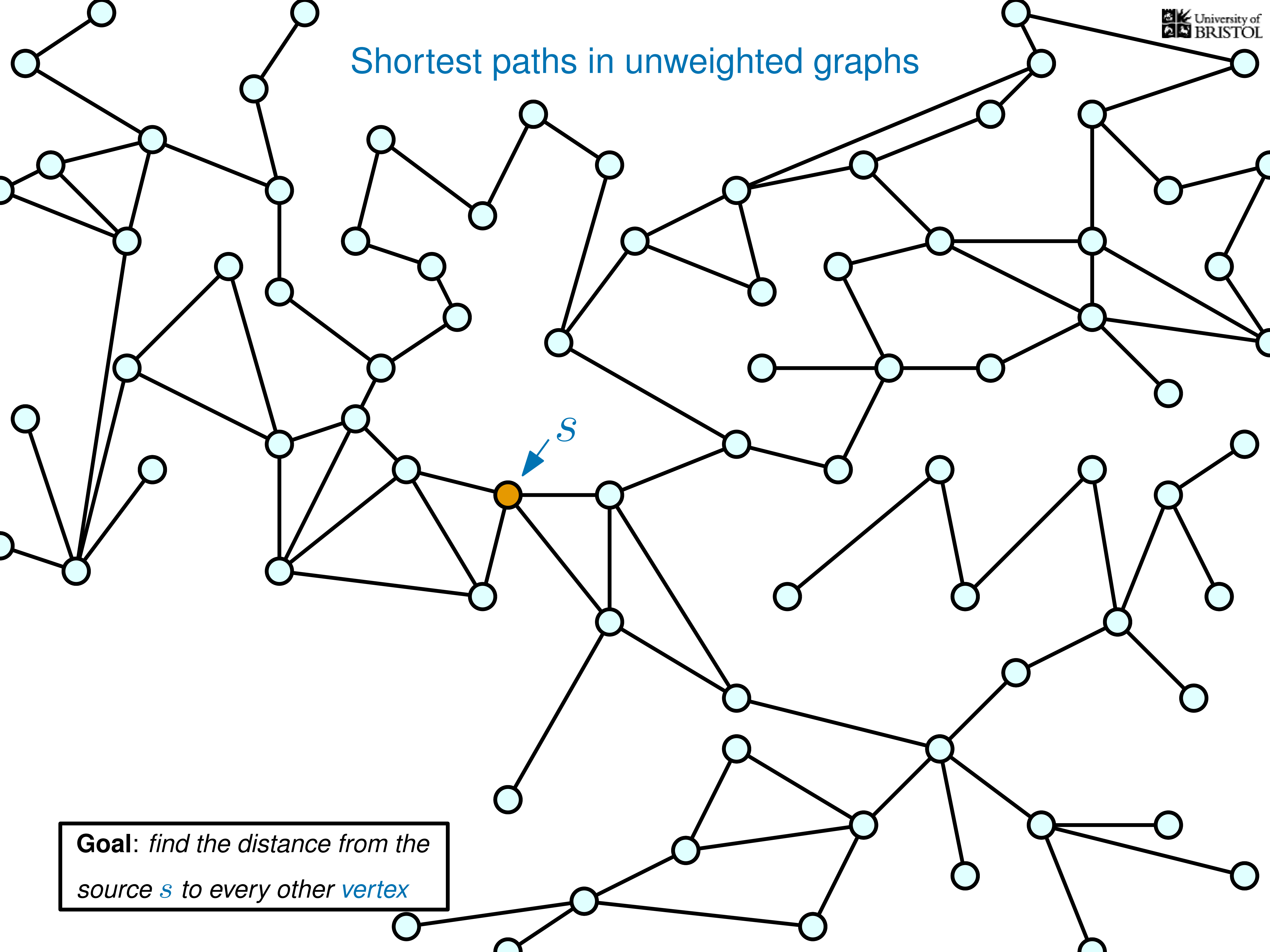
with a Stack the algorithm is called

Depth First Search

Applications

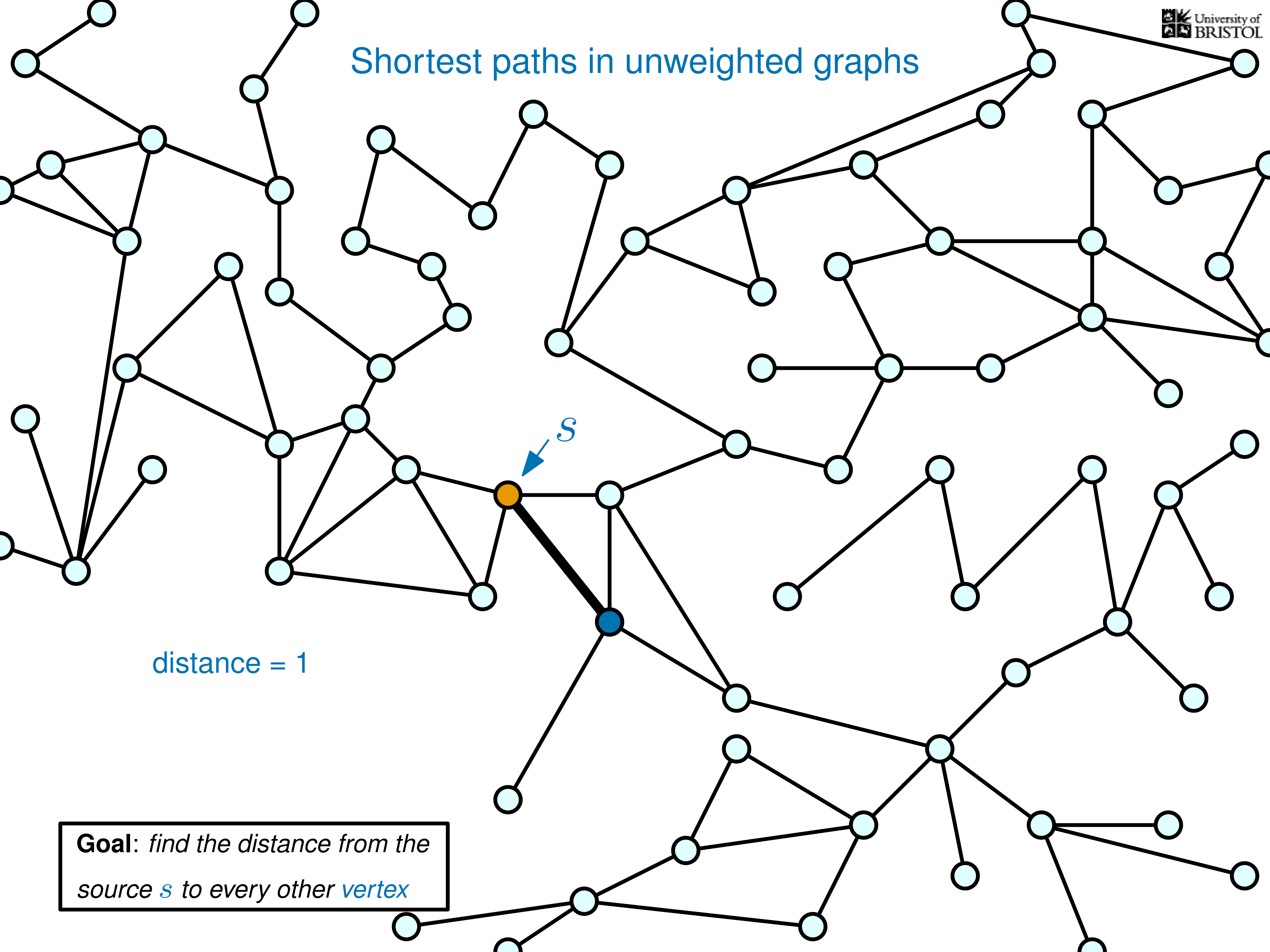
- Finding (strongly) connected components
- Topicologically sorting a Directed Acyclic Graph
- Testing for planarity

Shortest paths in unweighted graphs



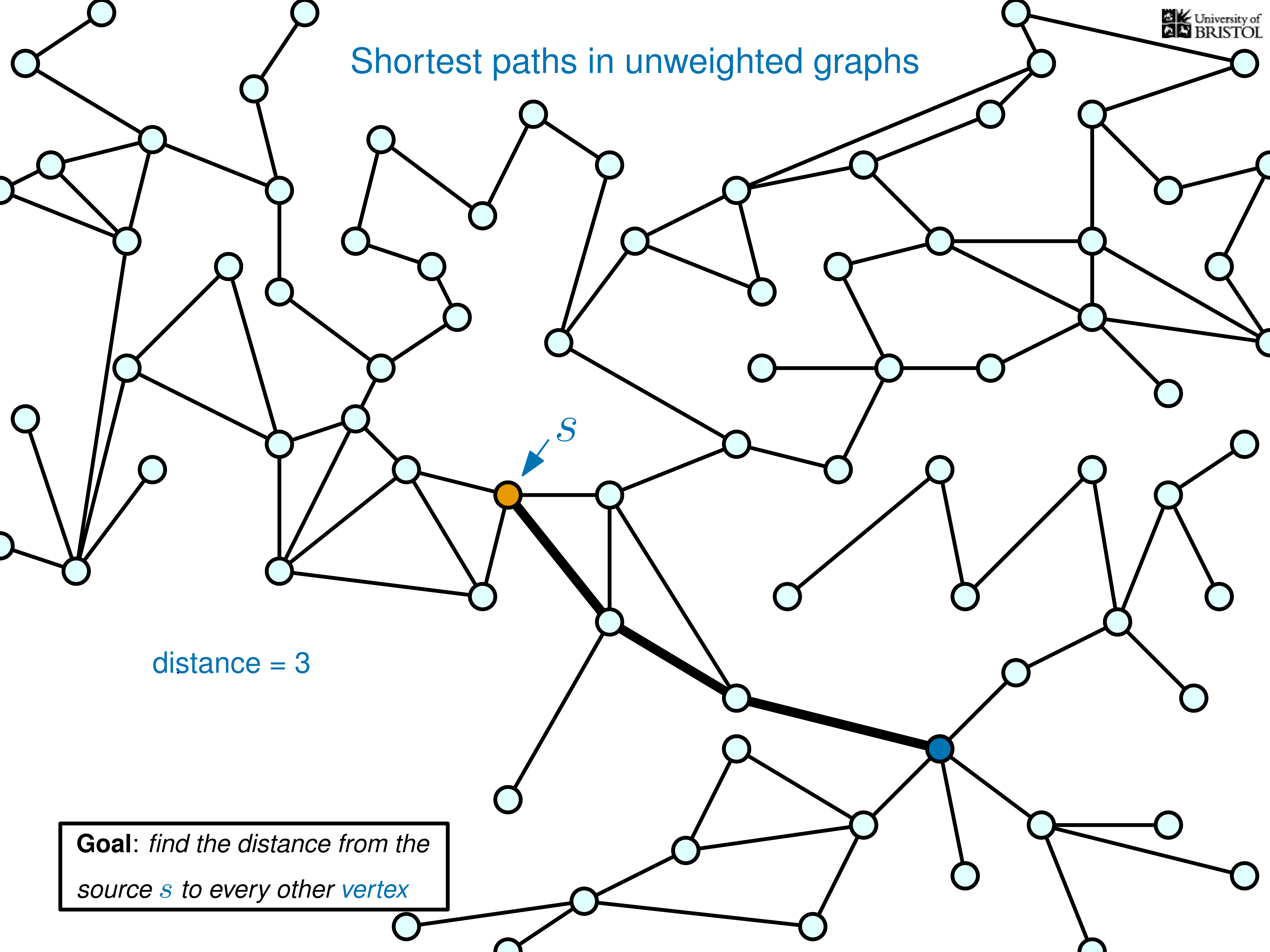
Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



Goal: *find the distance from the source s to every other vertex*

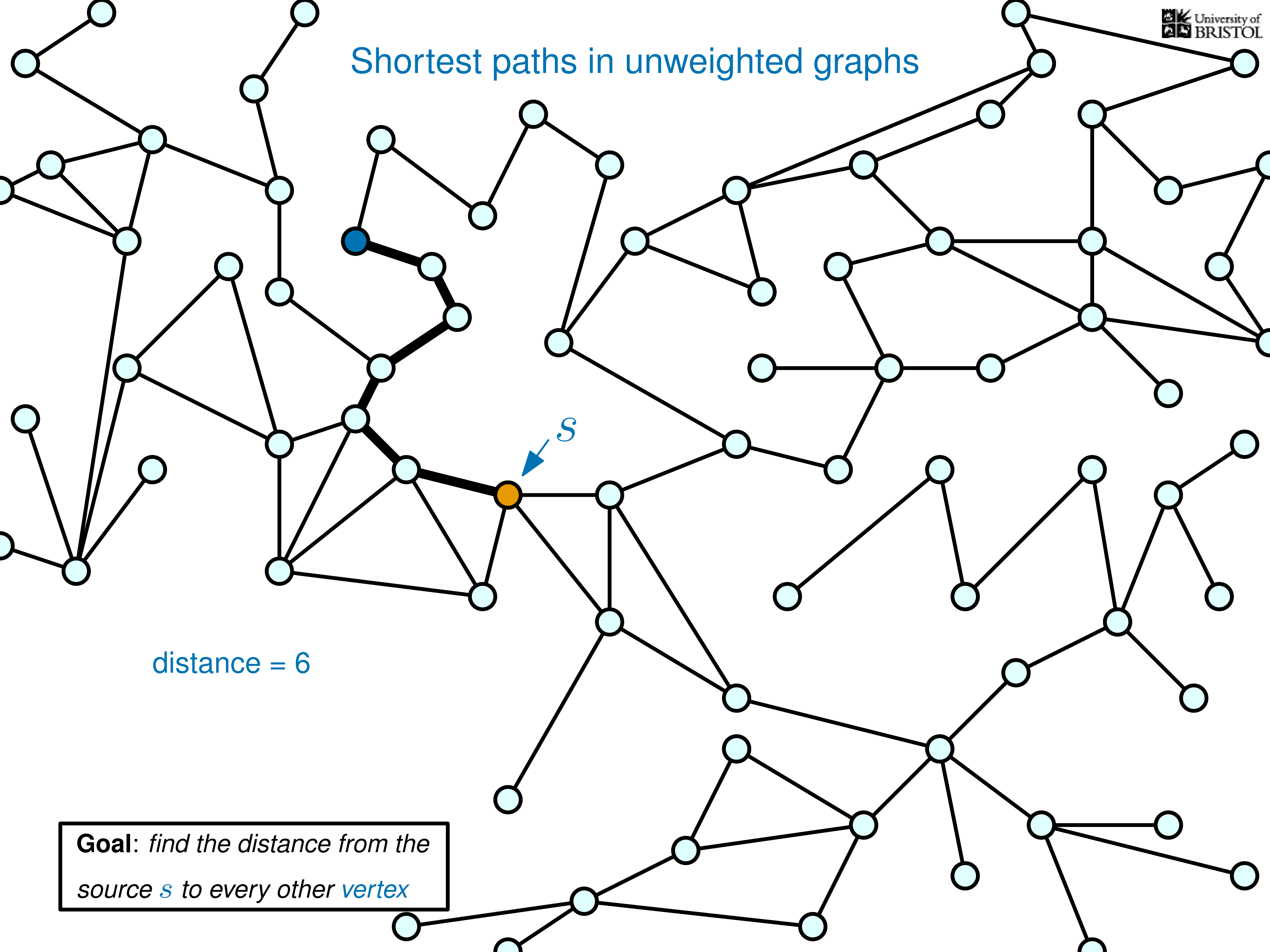
Shortest paths in unweighted graphs



distance = 3

Goal: *find the distance from the source s to every other vertex*

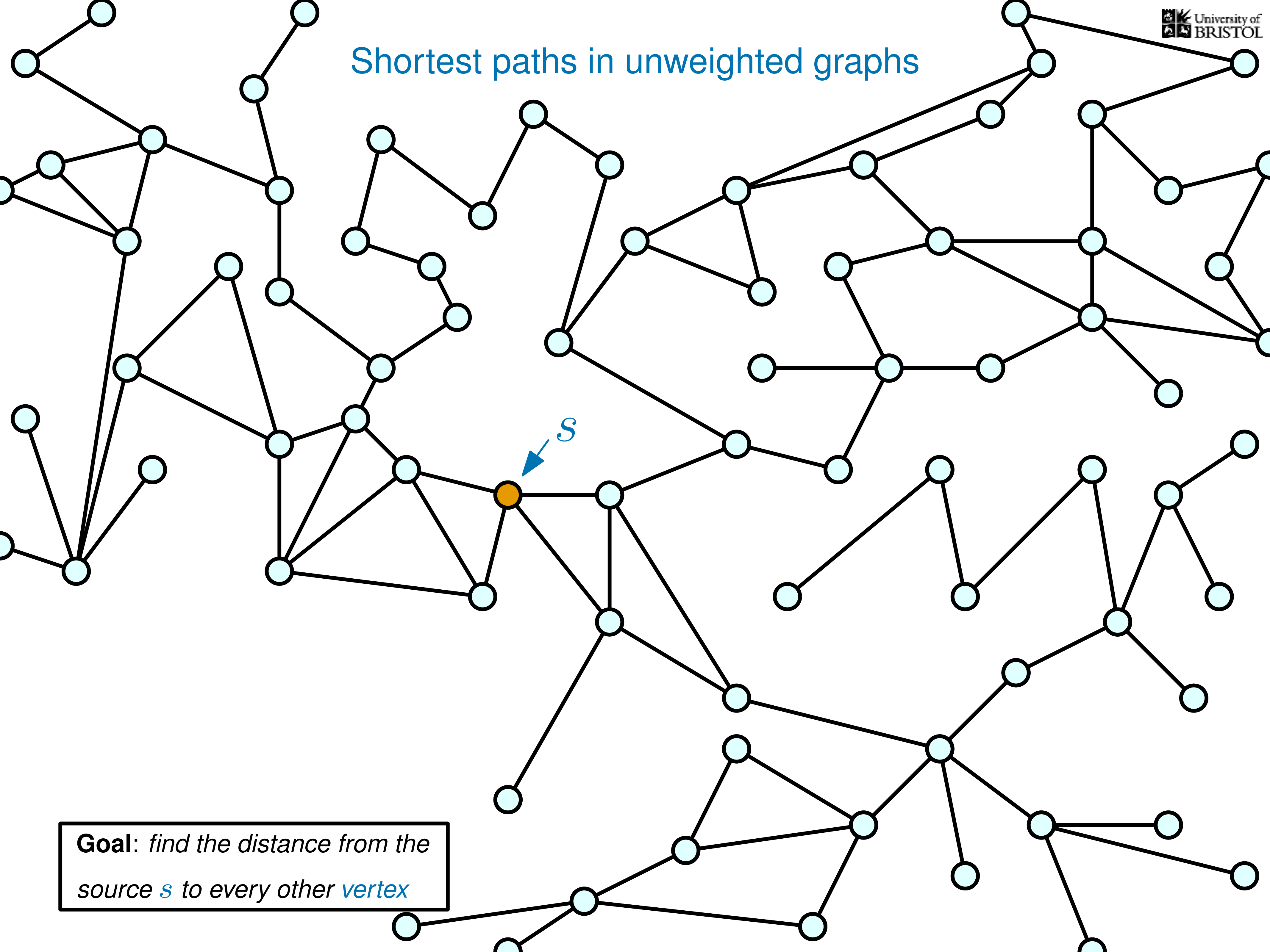
Shortest paths in unweighted graphs



distance = 6

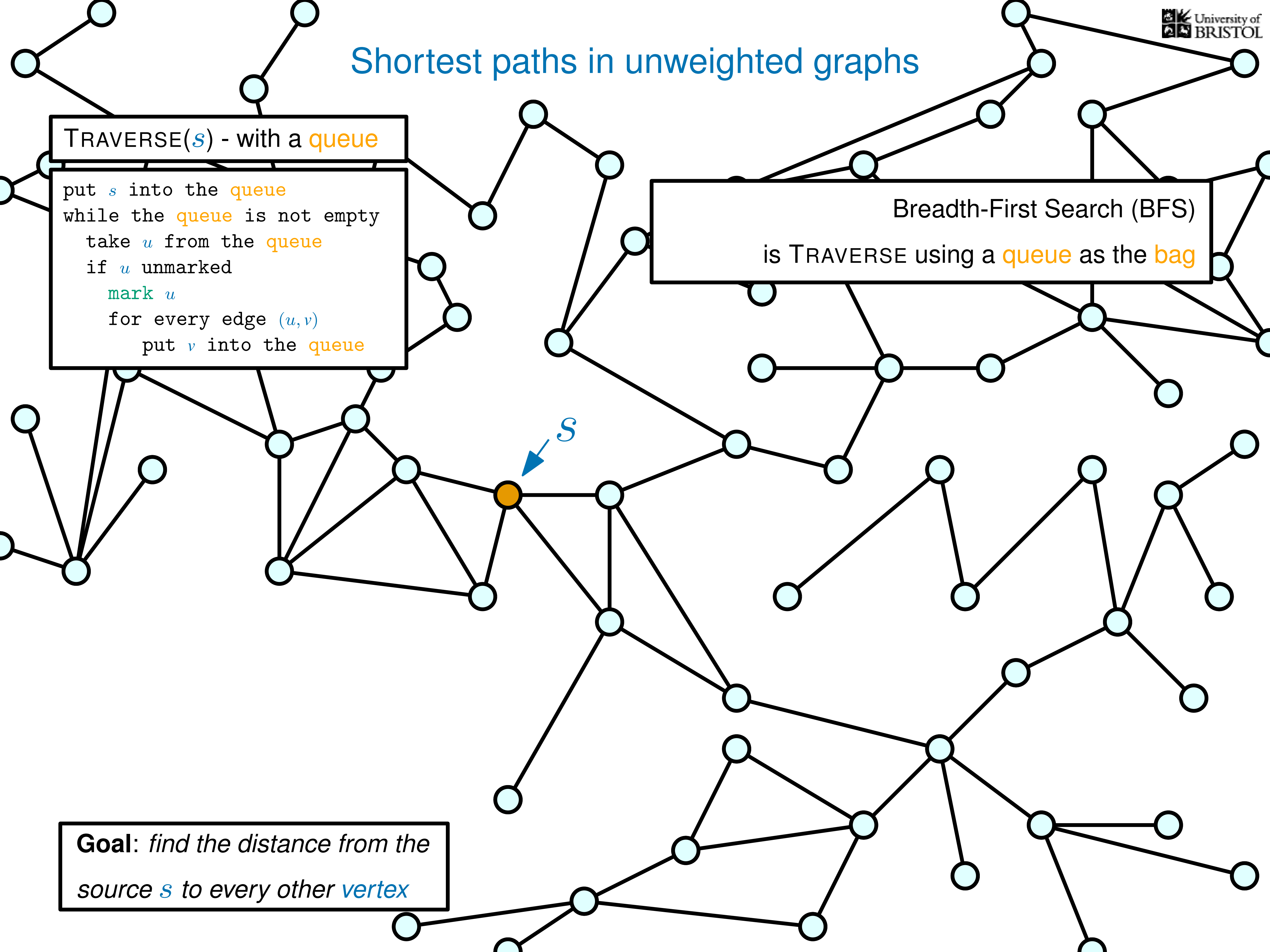
Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

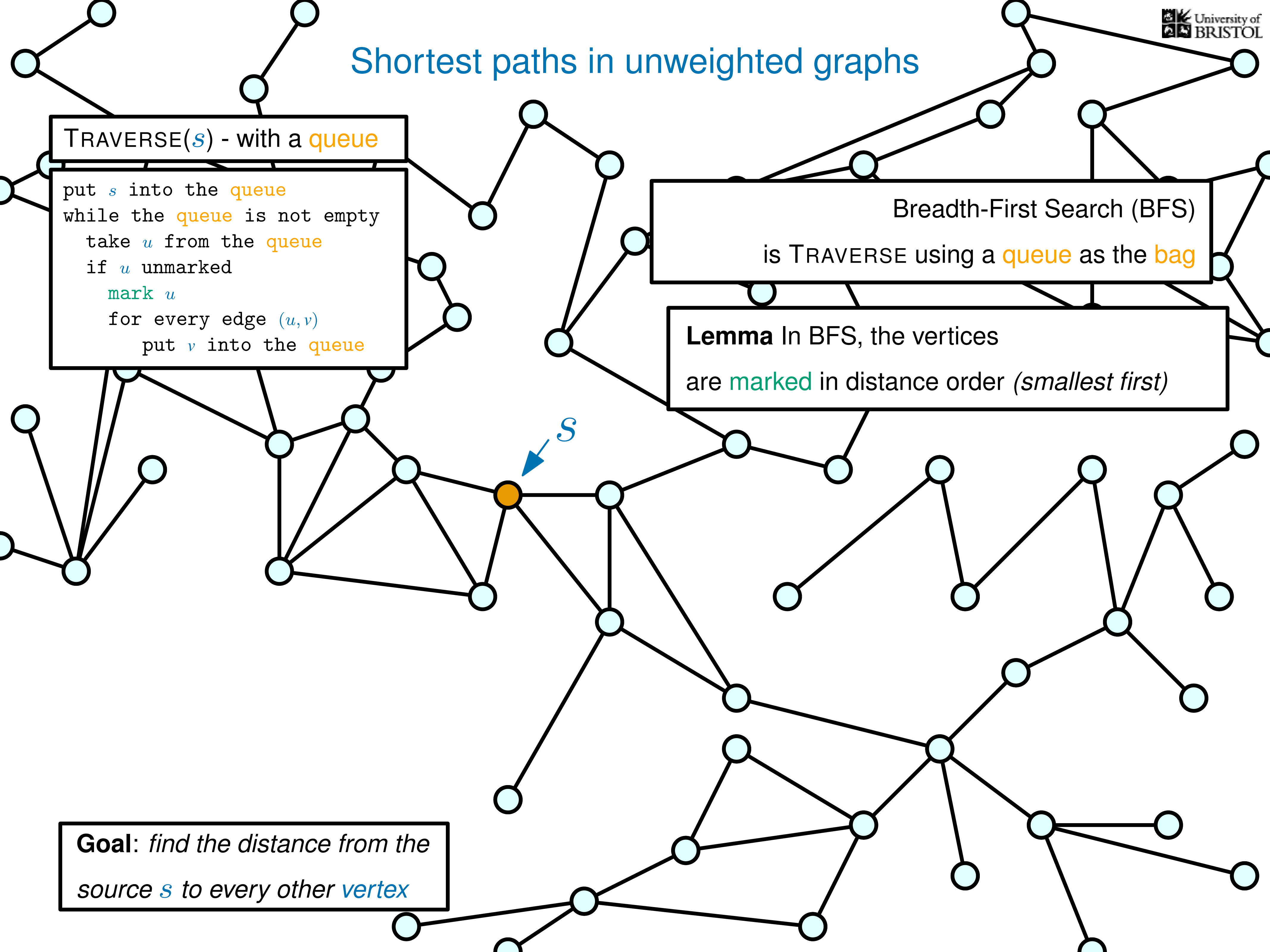
```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

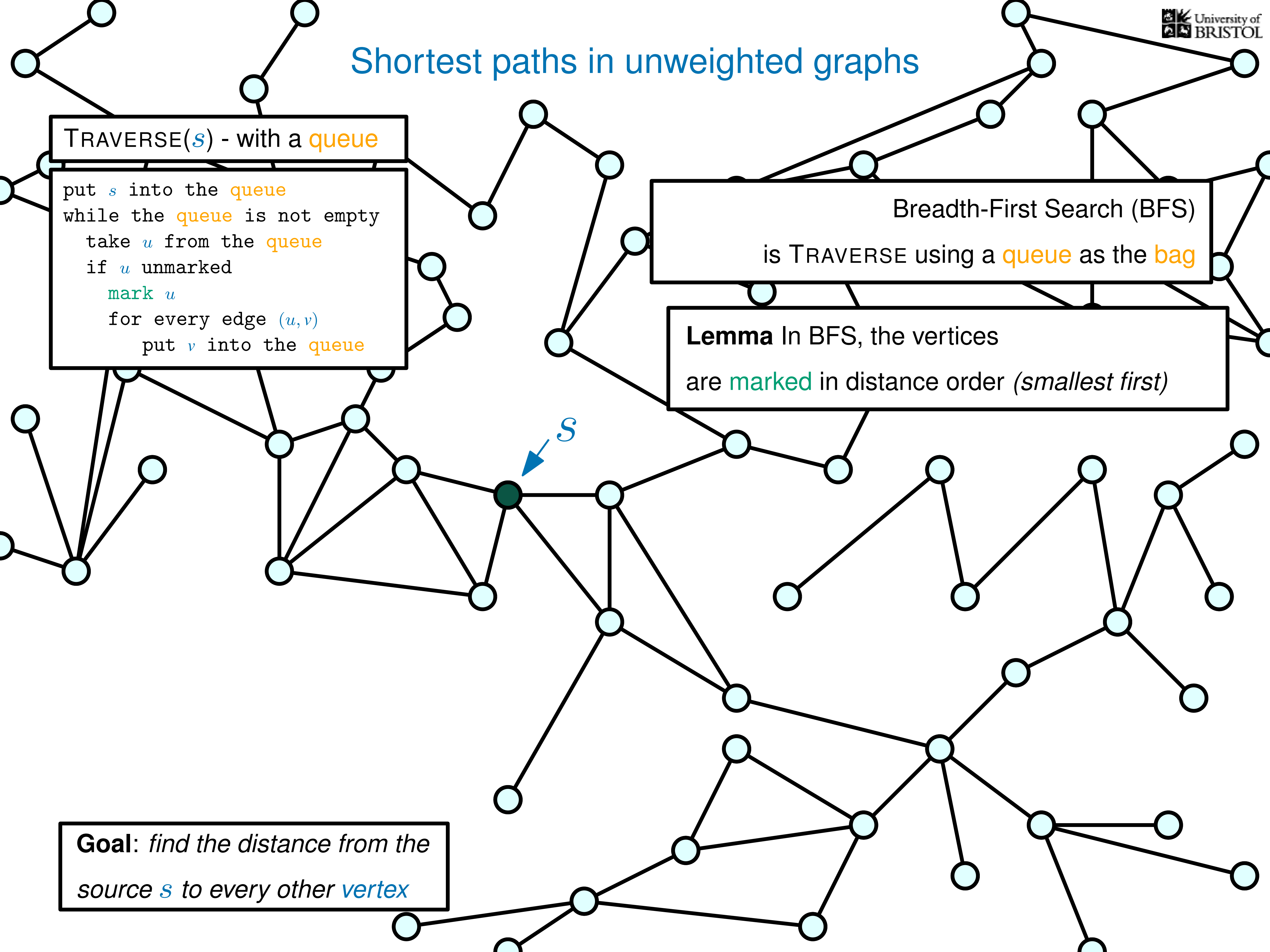
Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

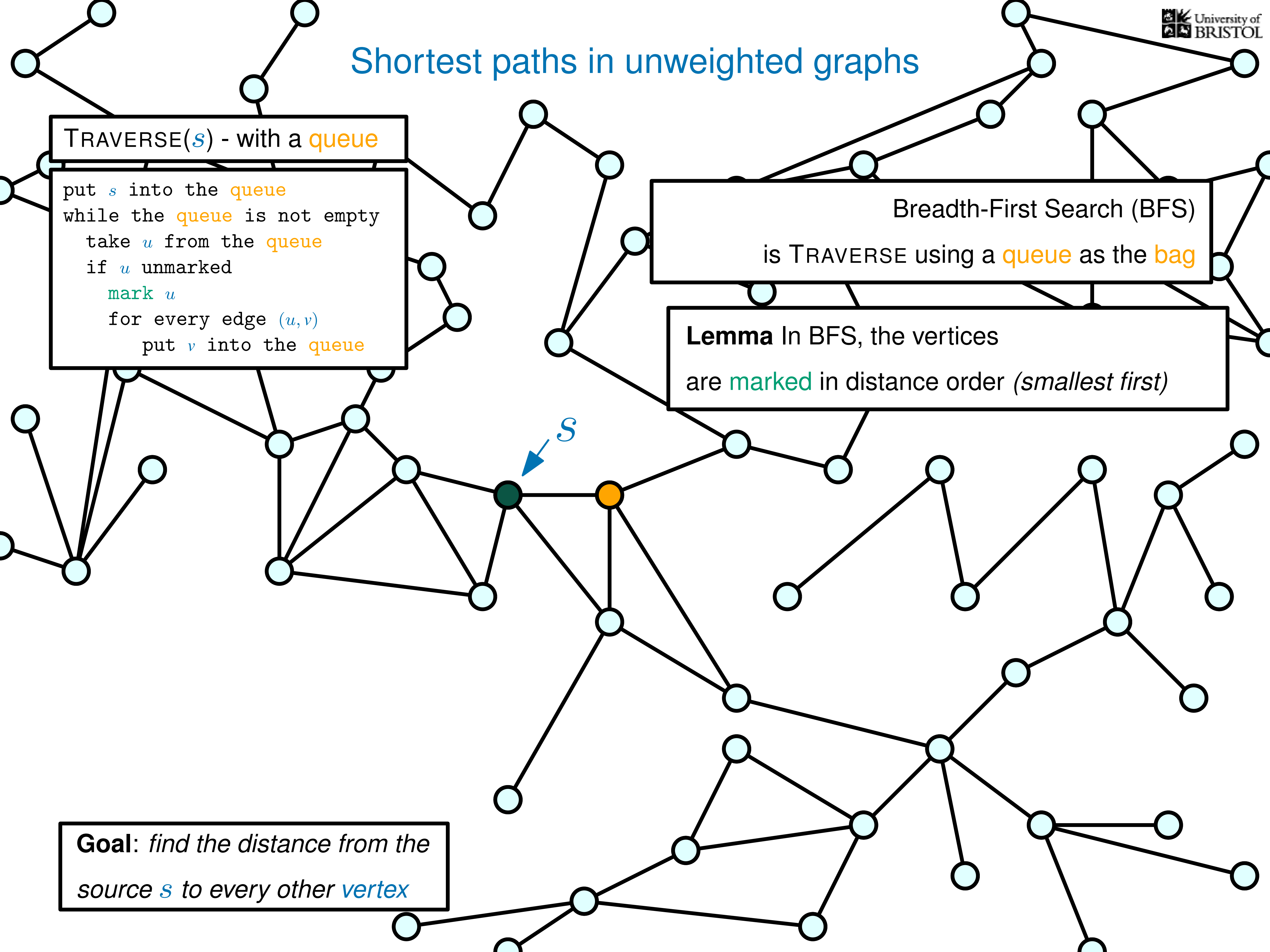
is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

Goal: *find the distance from the source* s *to every other vertex*

University of BRISTOL

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices

are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

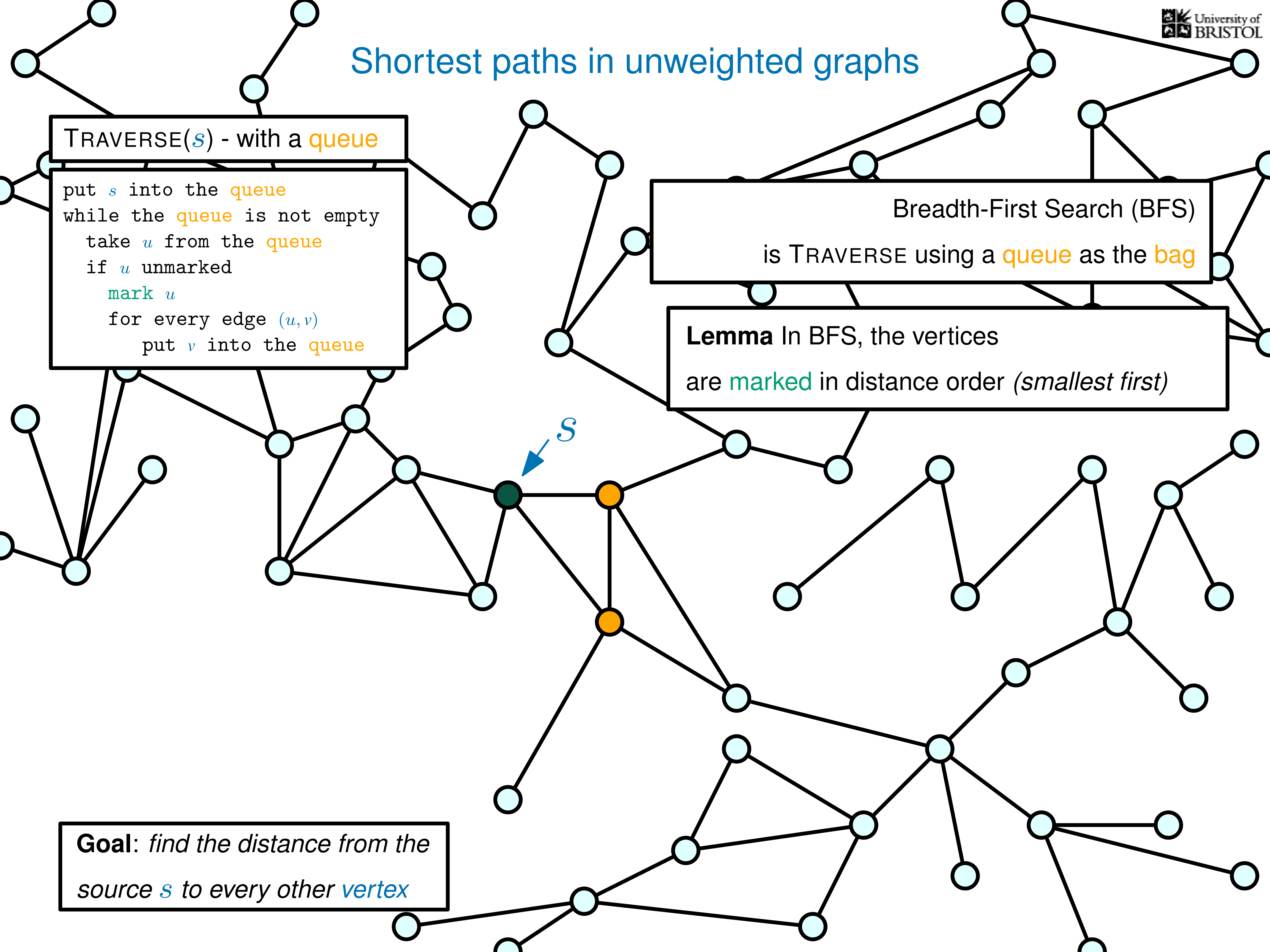
TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Breadth-First Search (BFS)

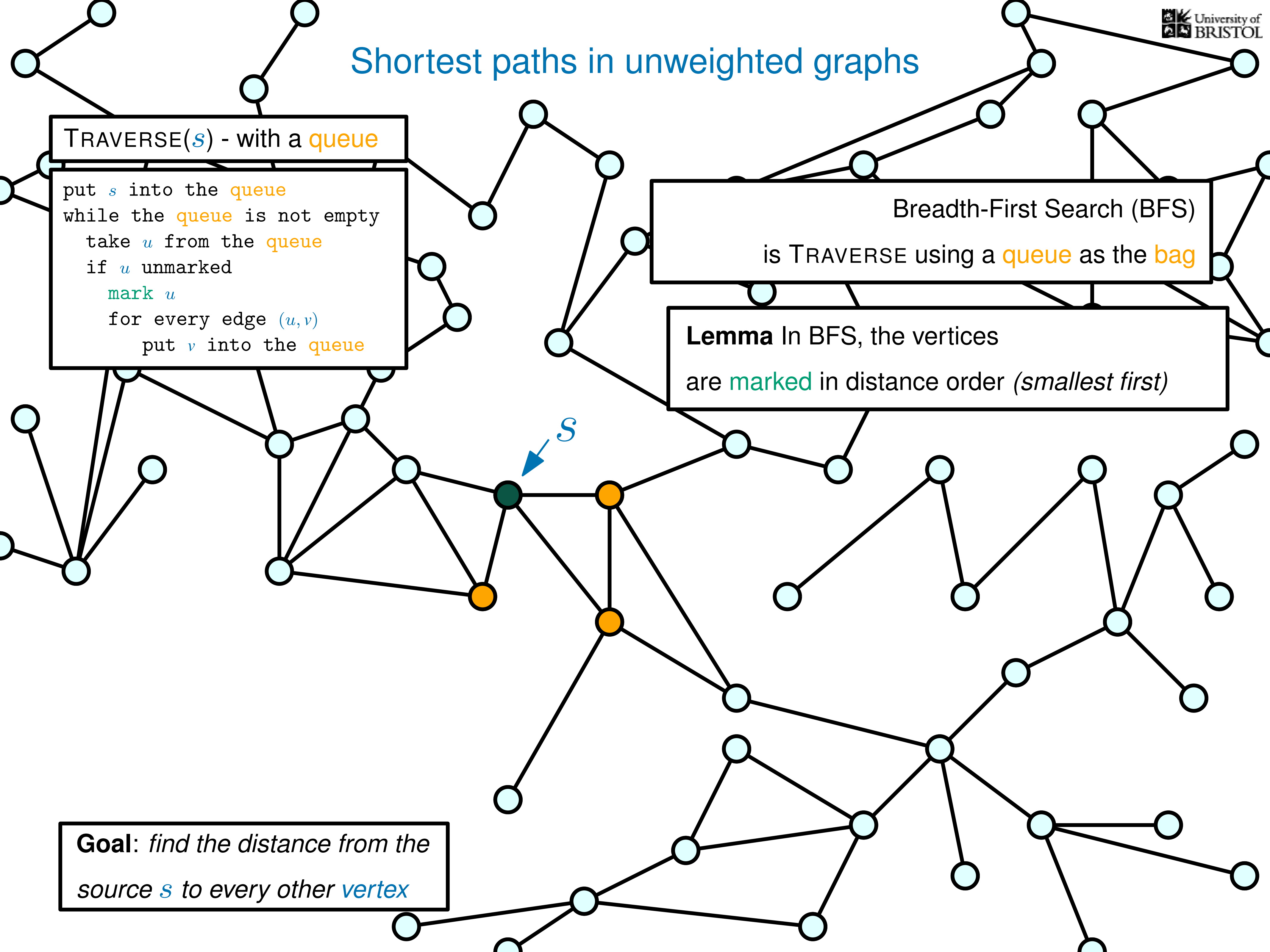
is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*



Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

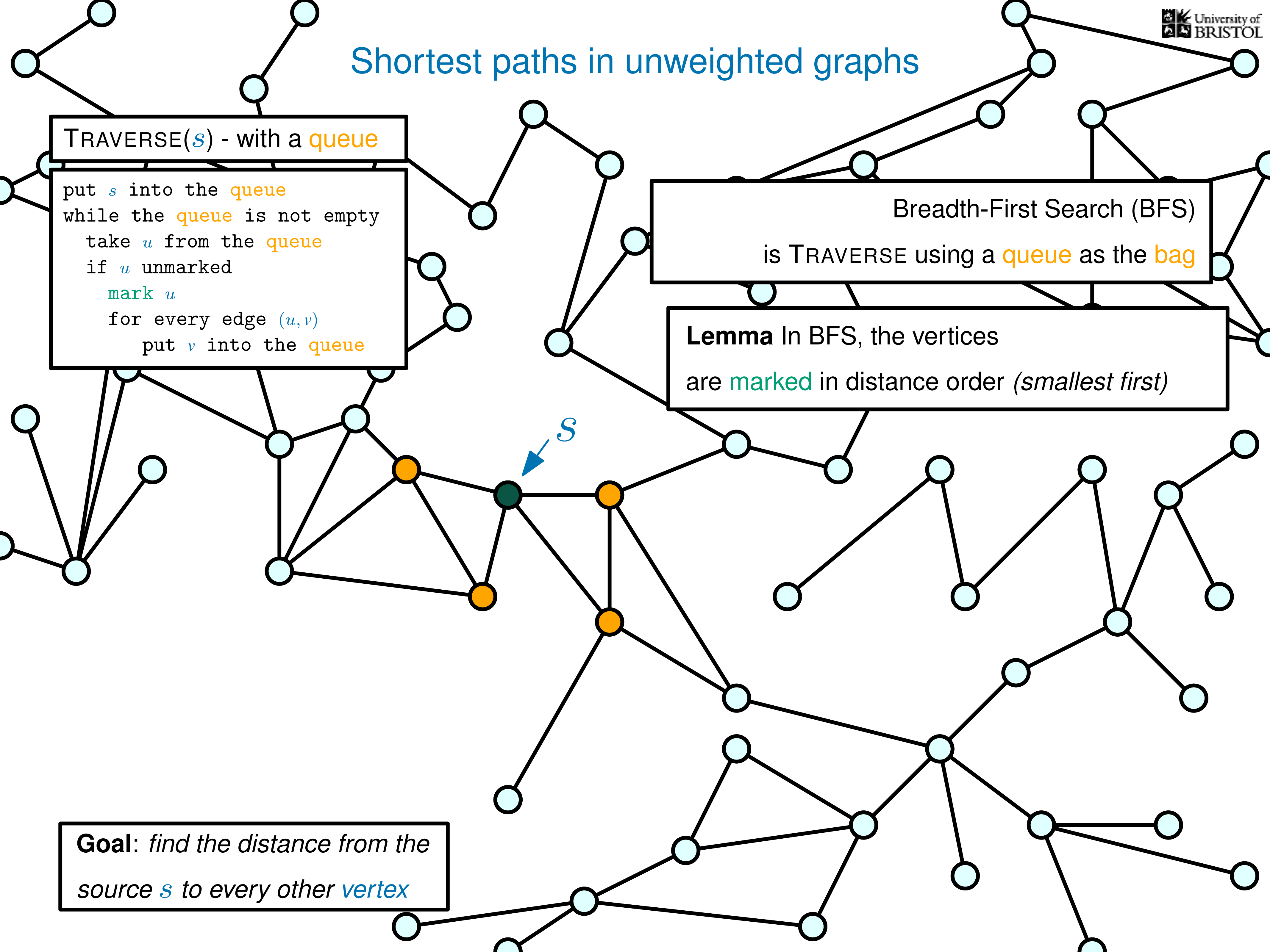
Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

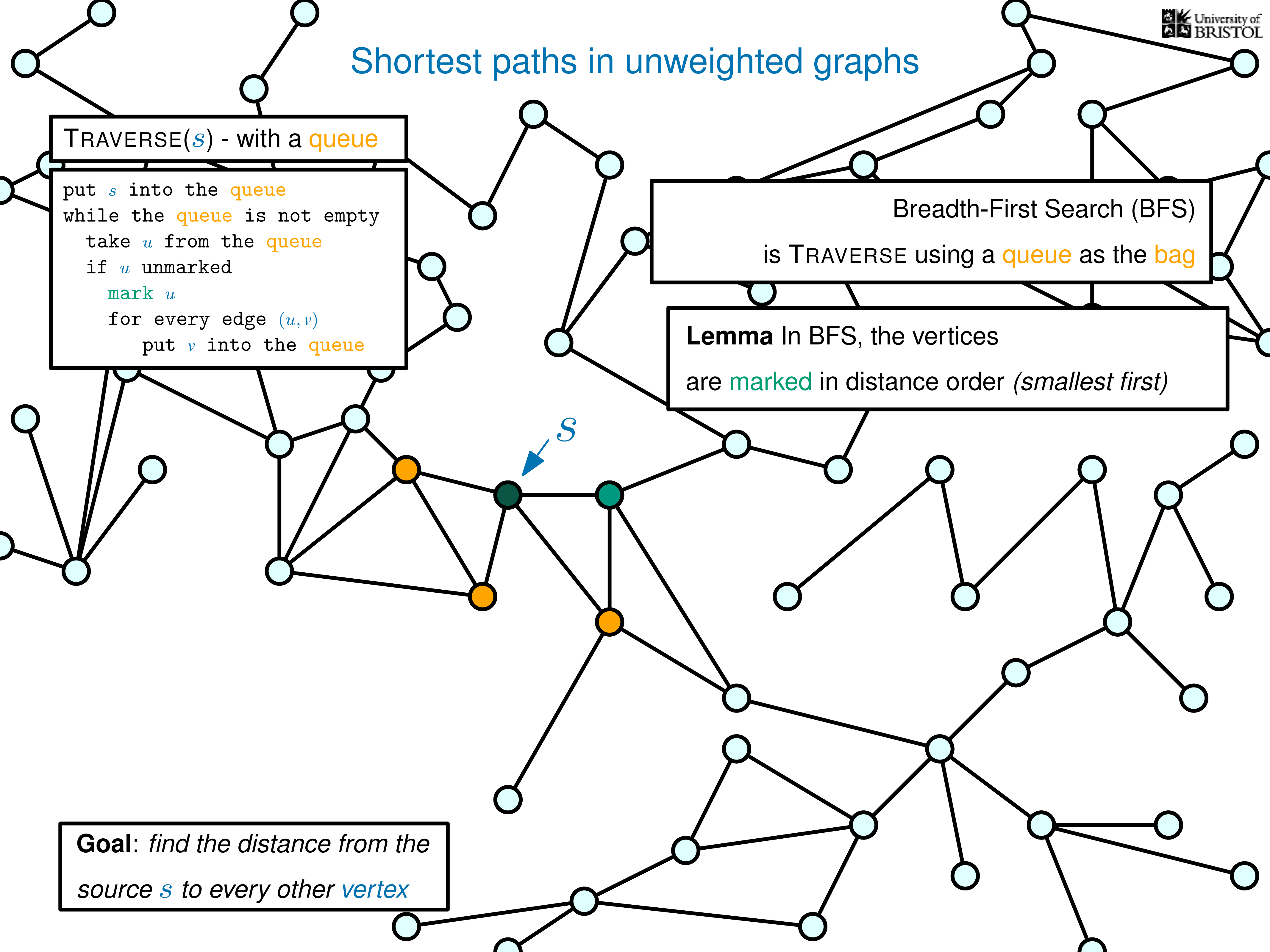
is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

University of BRISTOL

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices

are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

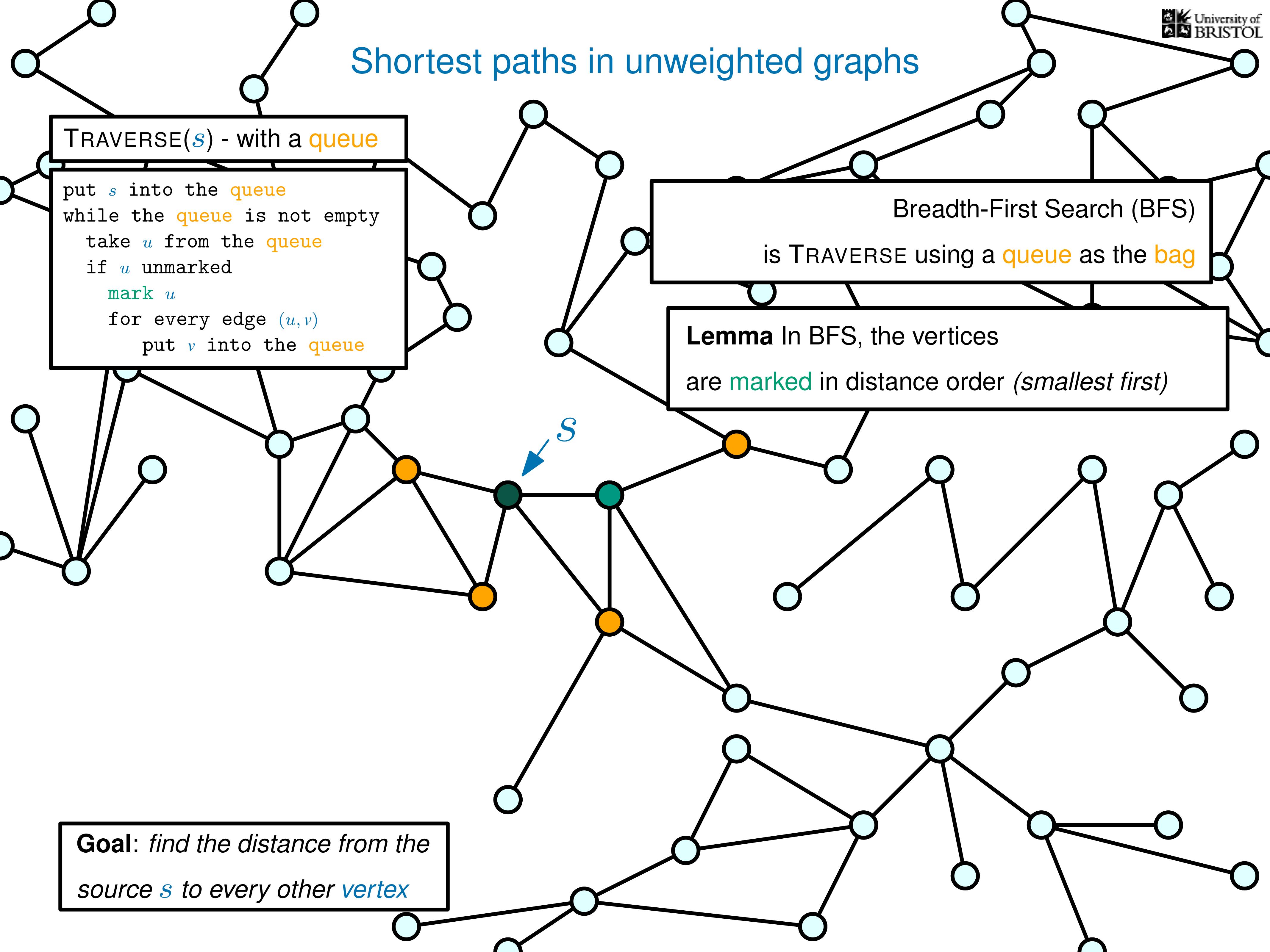
TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*



Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

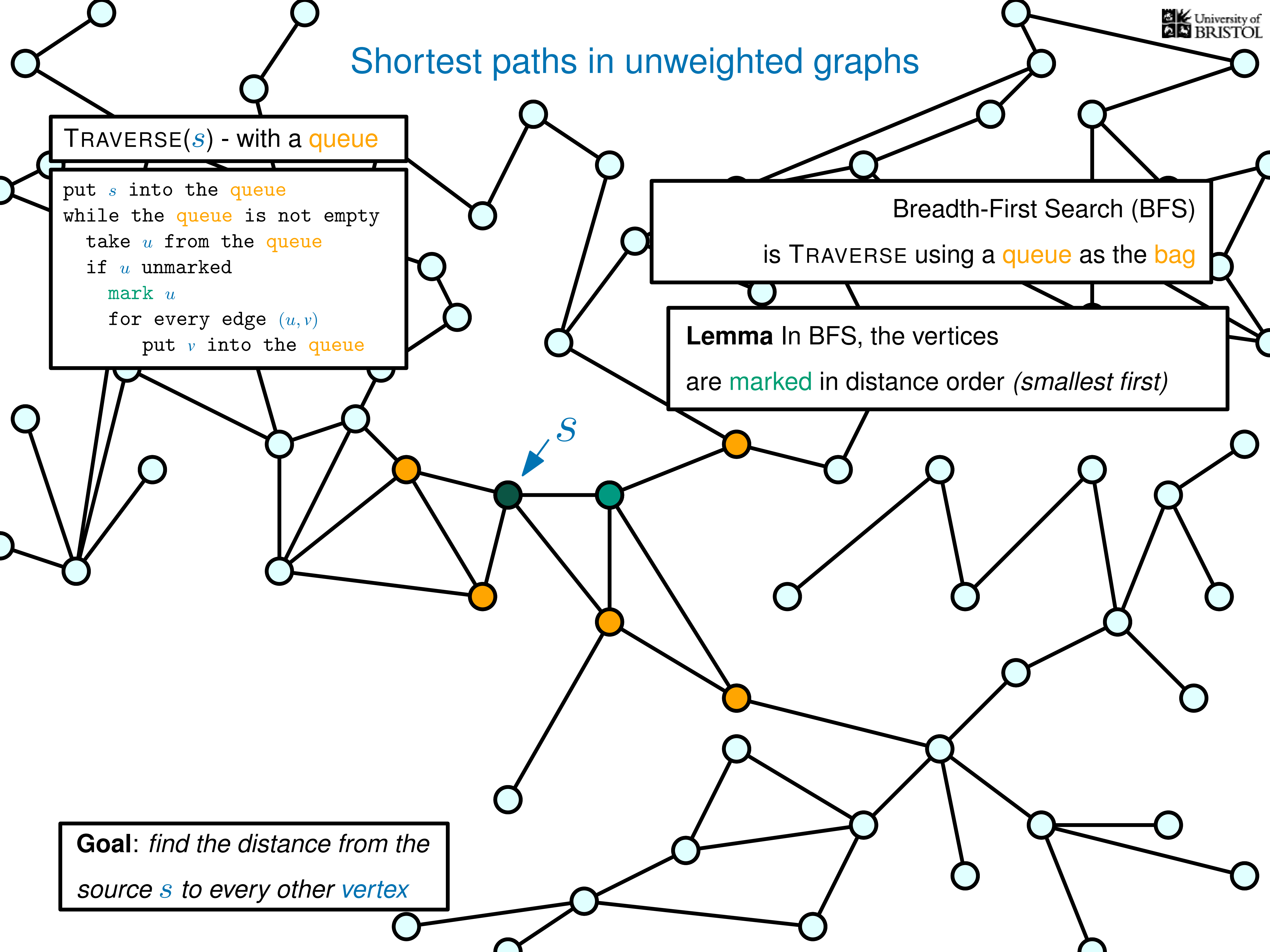
Traverse(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

is Traverse using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*



Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

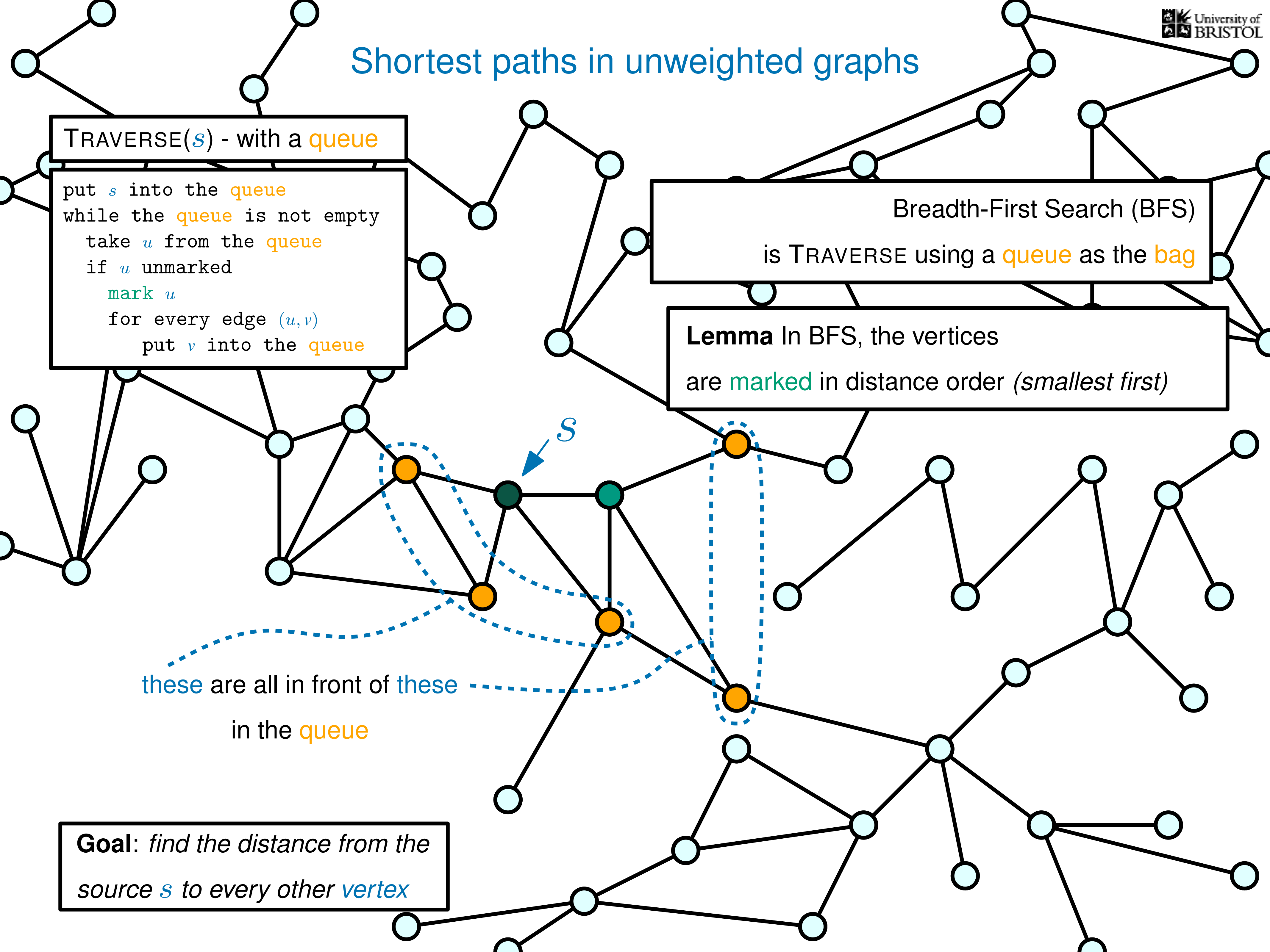
TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*



Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

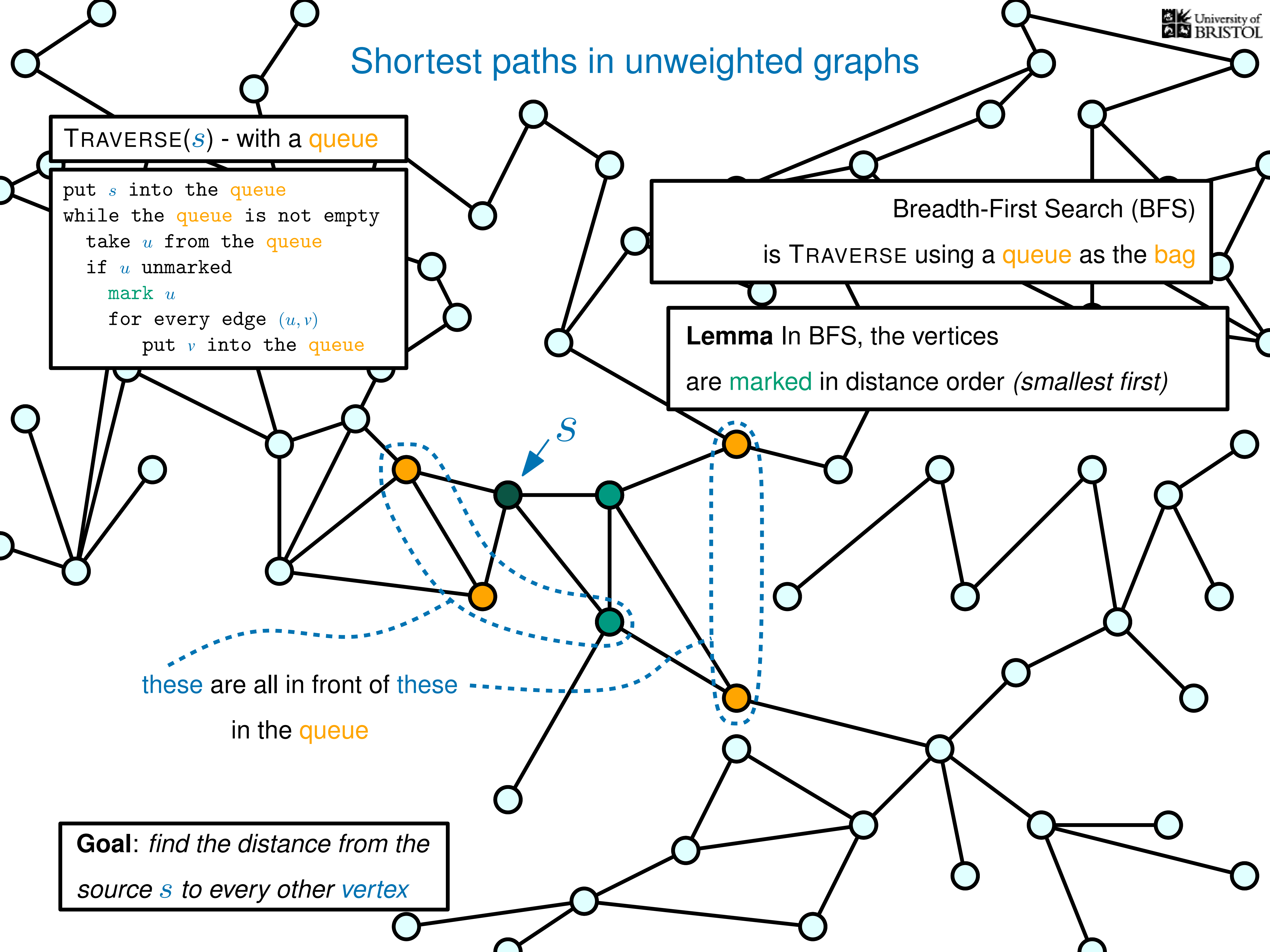
TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

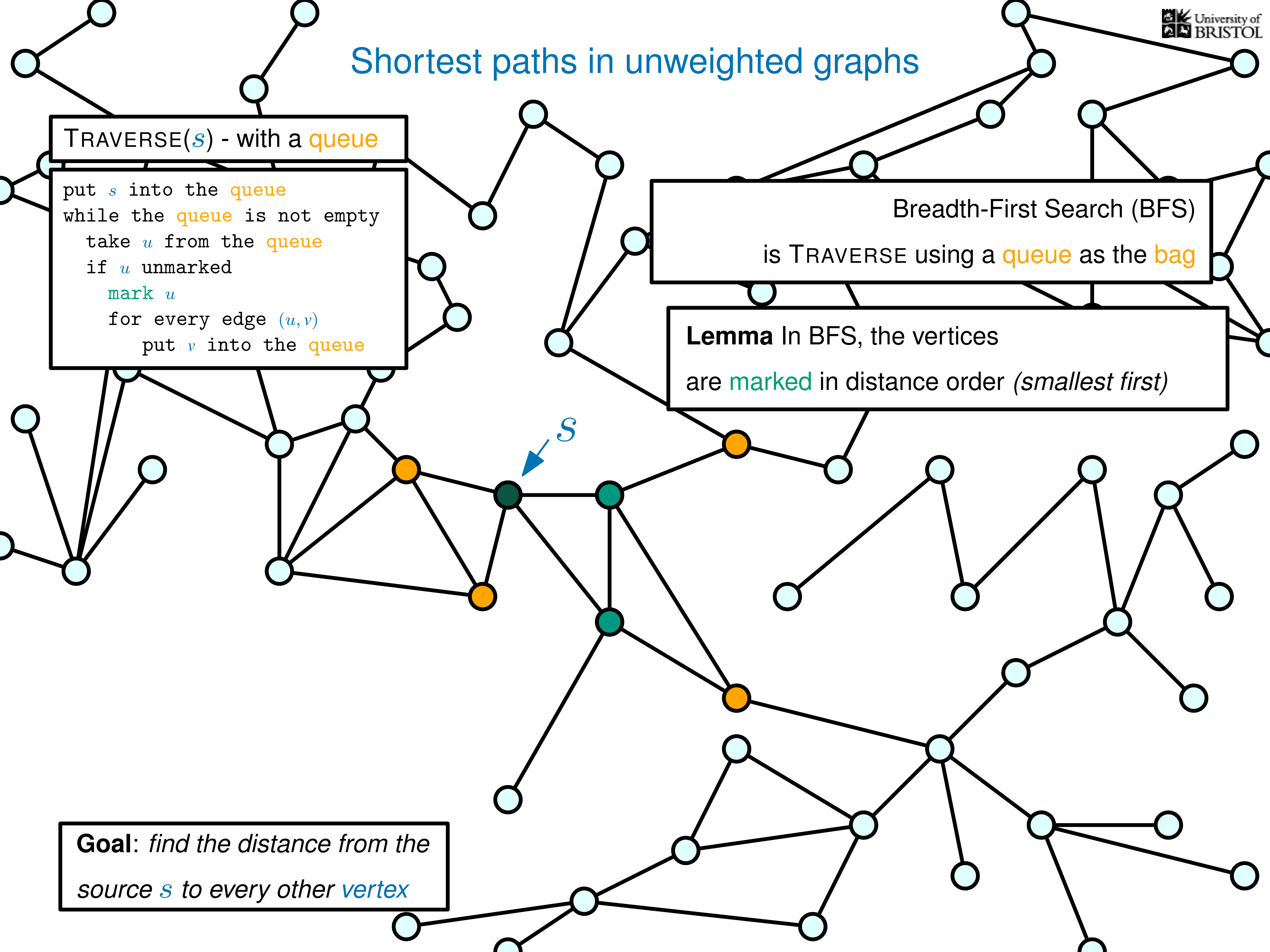
is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*



Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

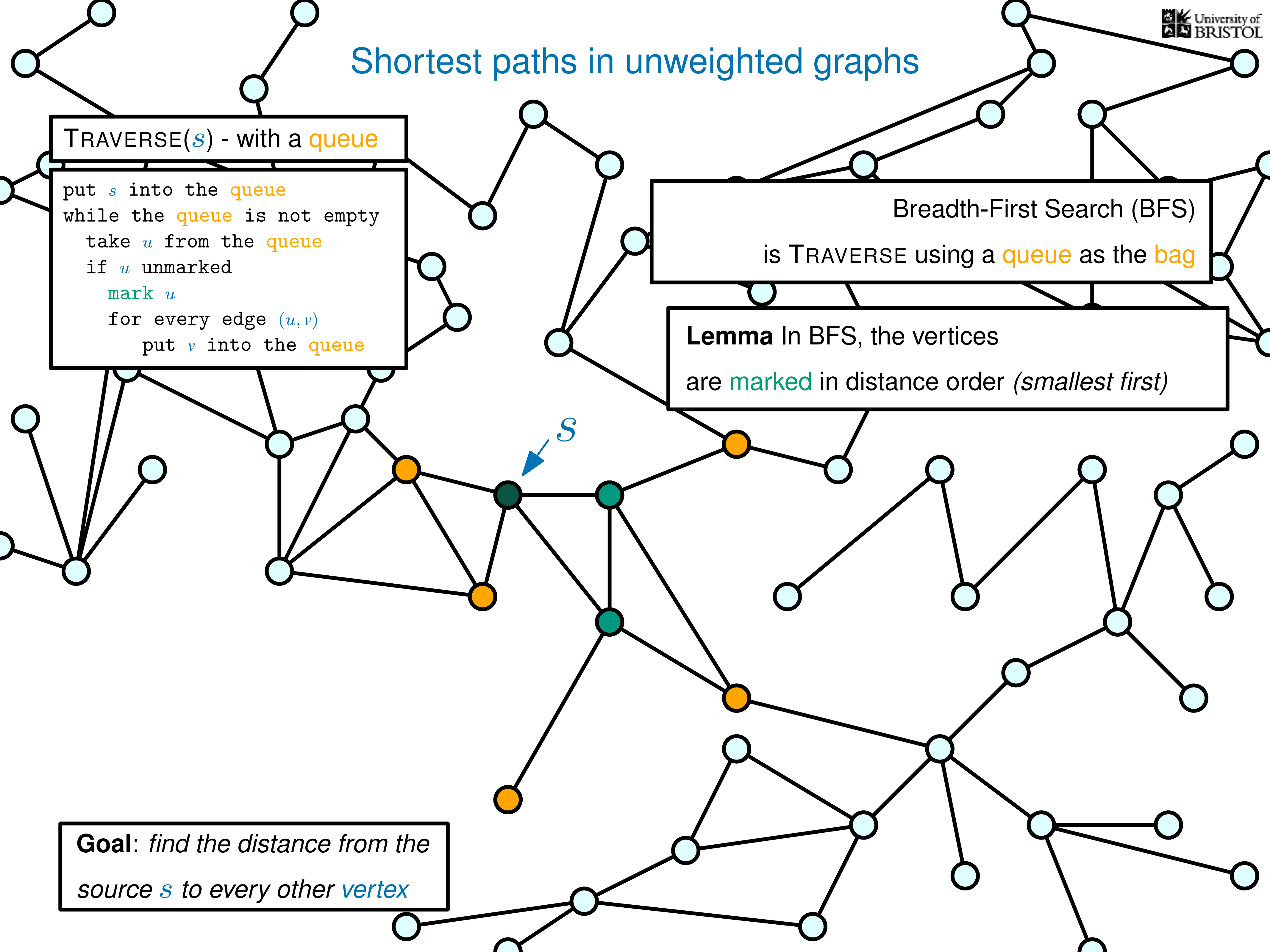
TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Breadth-First Search (BFS)

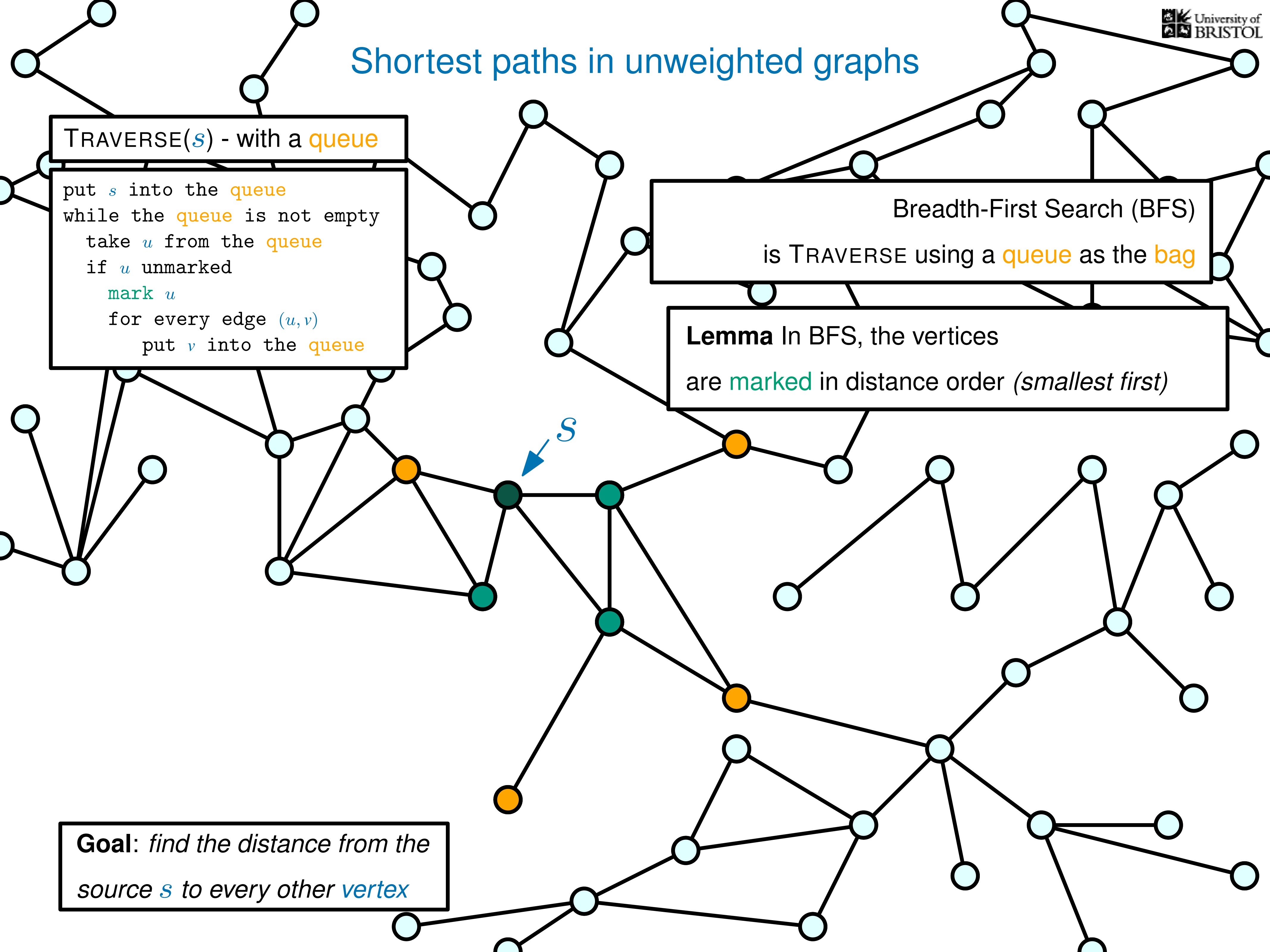
is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*



Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

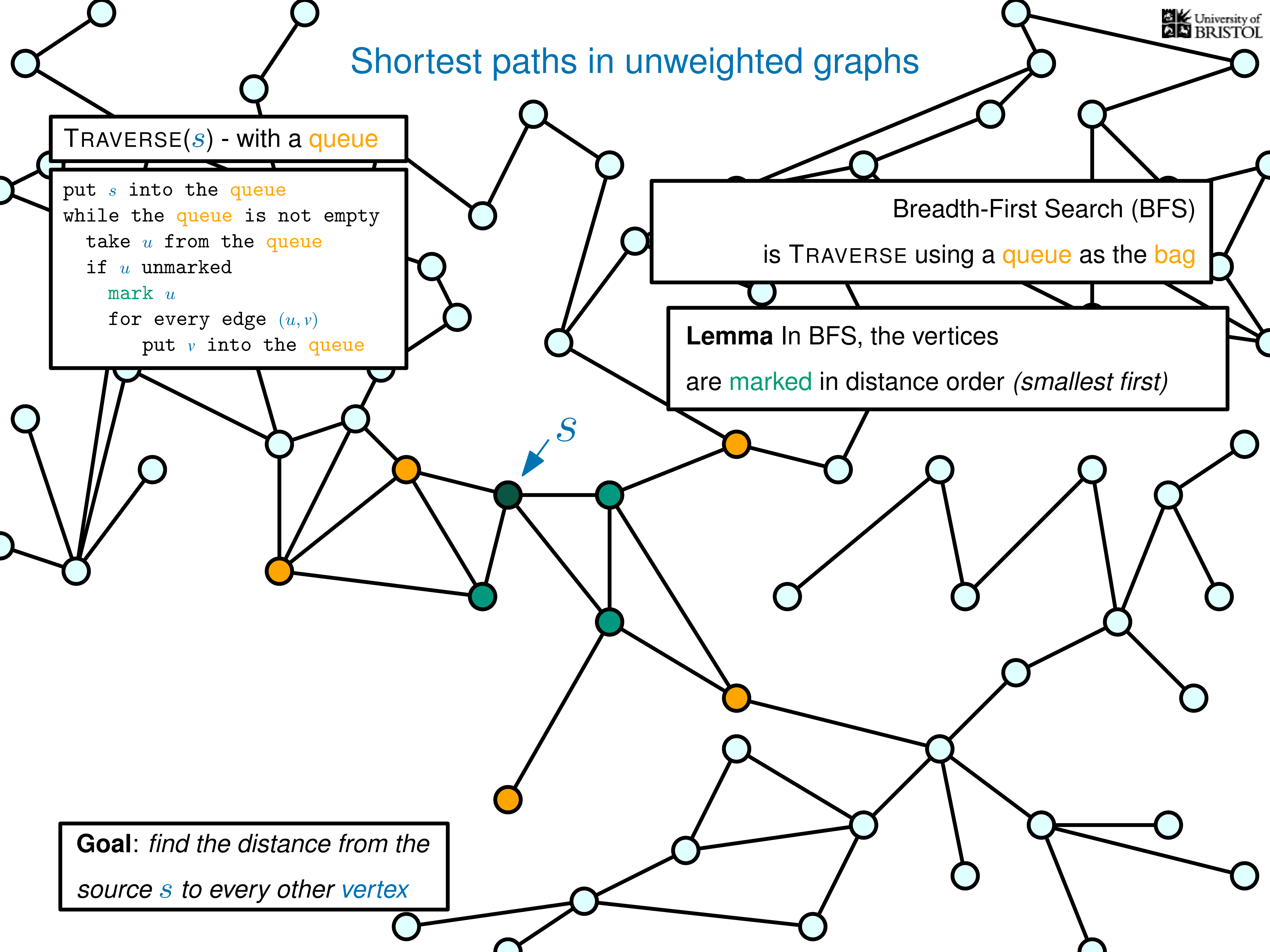
Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices

are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

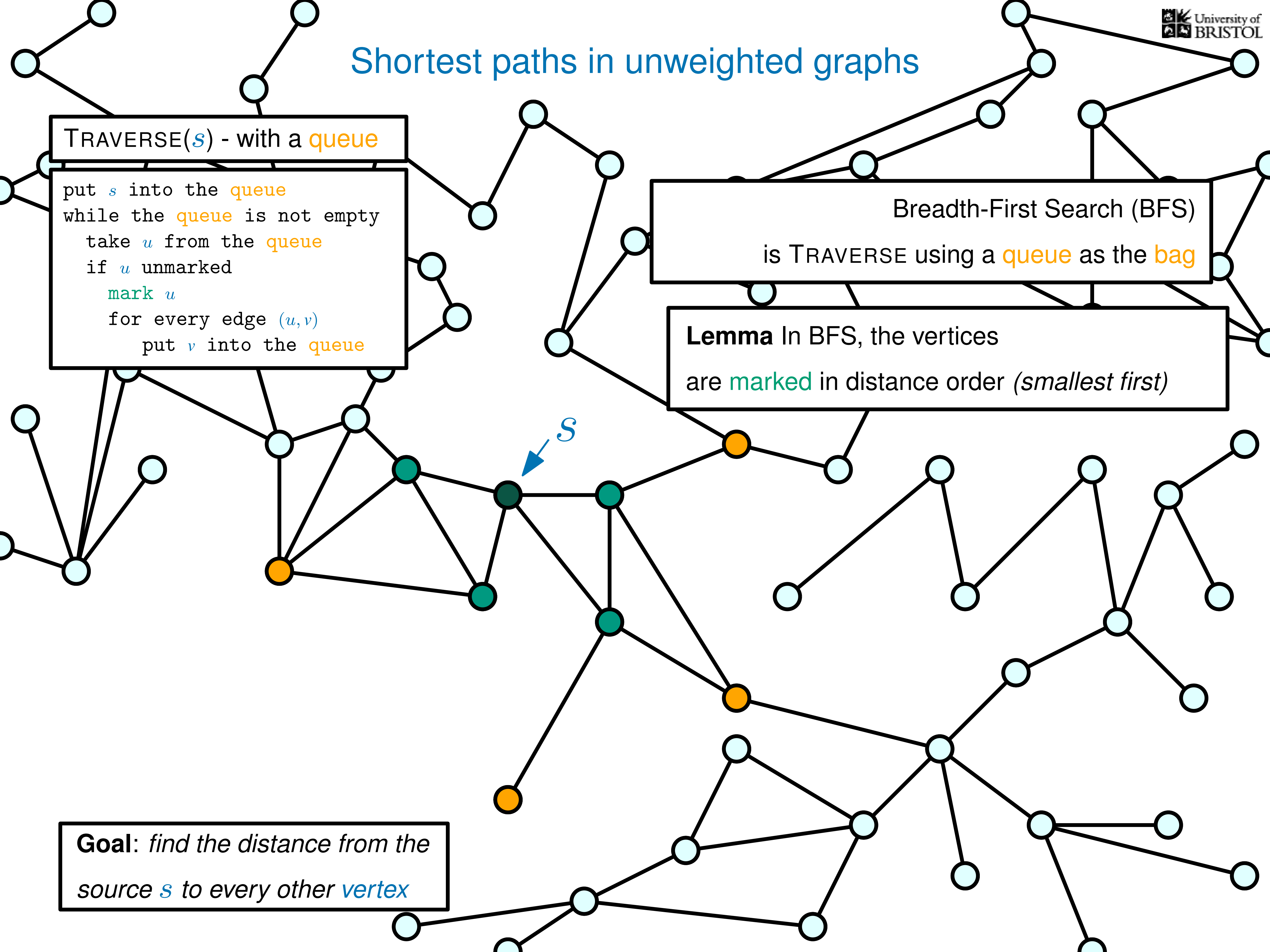
TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

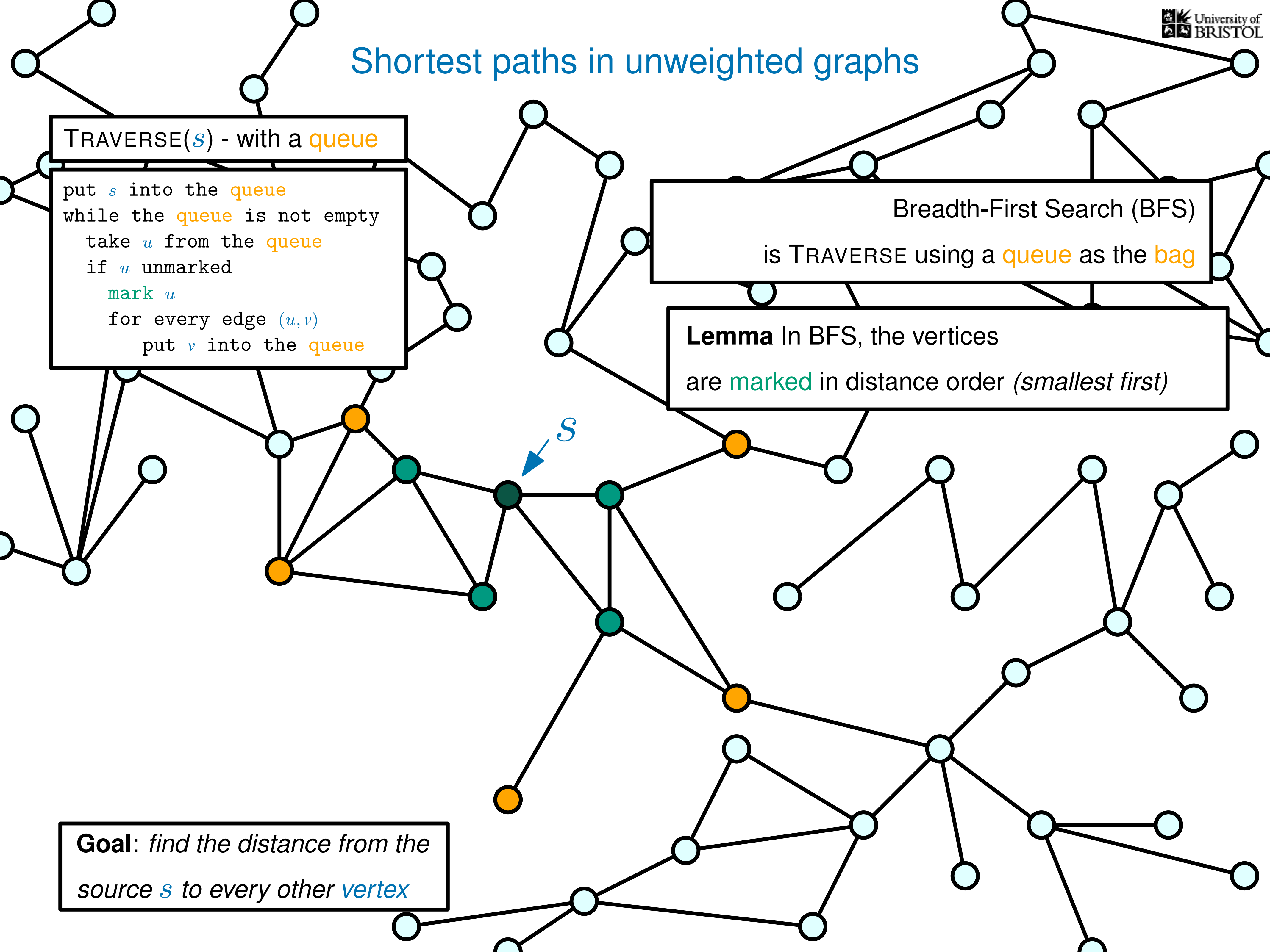
is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*



Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

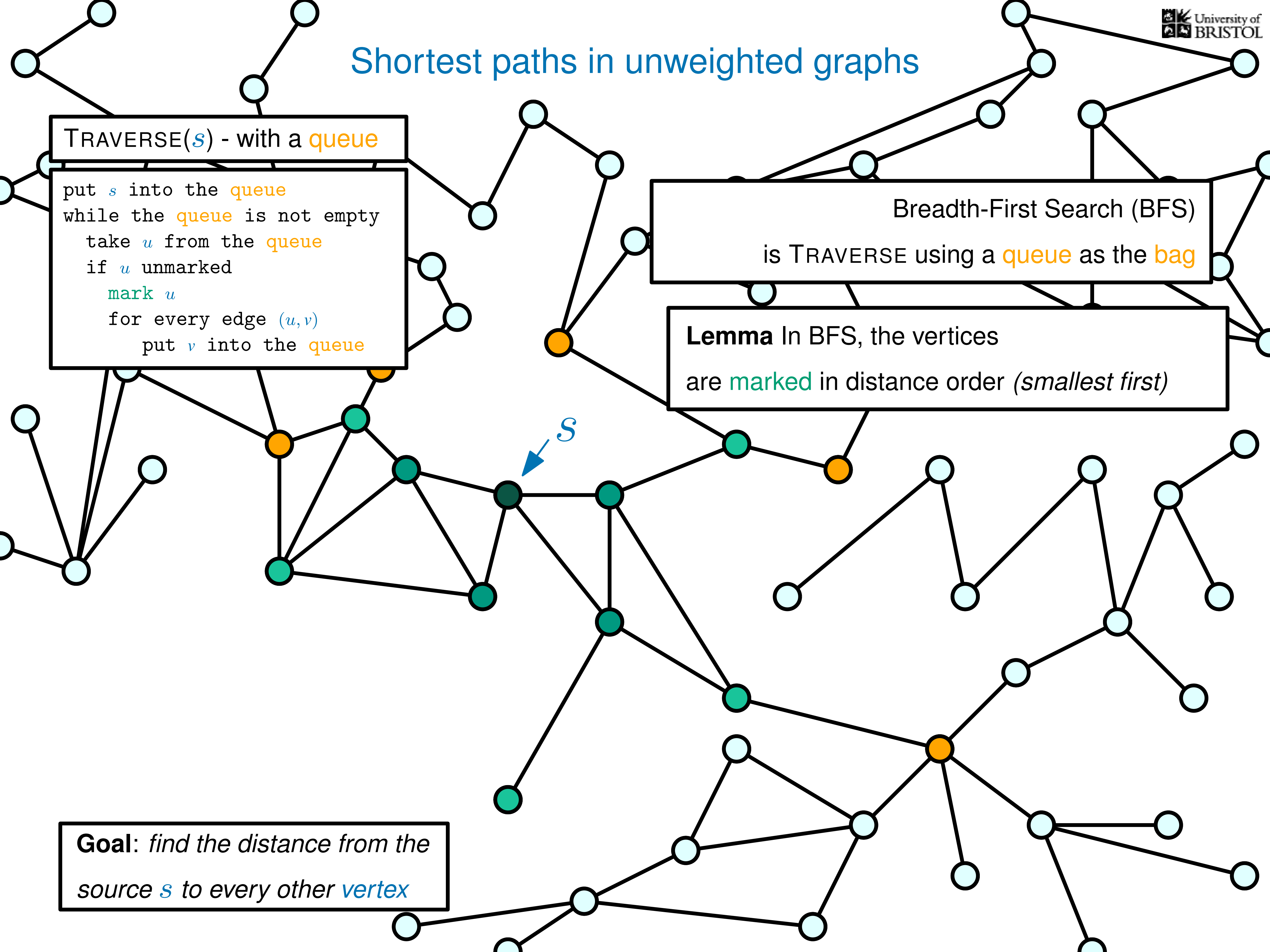
Lemma In BFS, the vertices

are marked in distance order *(smallest first)*

Goal: *find the distance from the*

source s to every other vertex

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

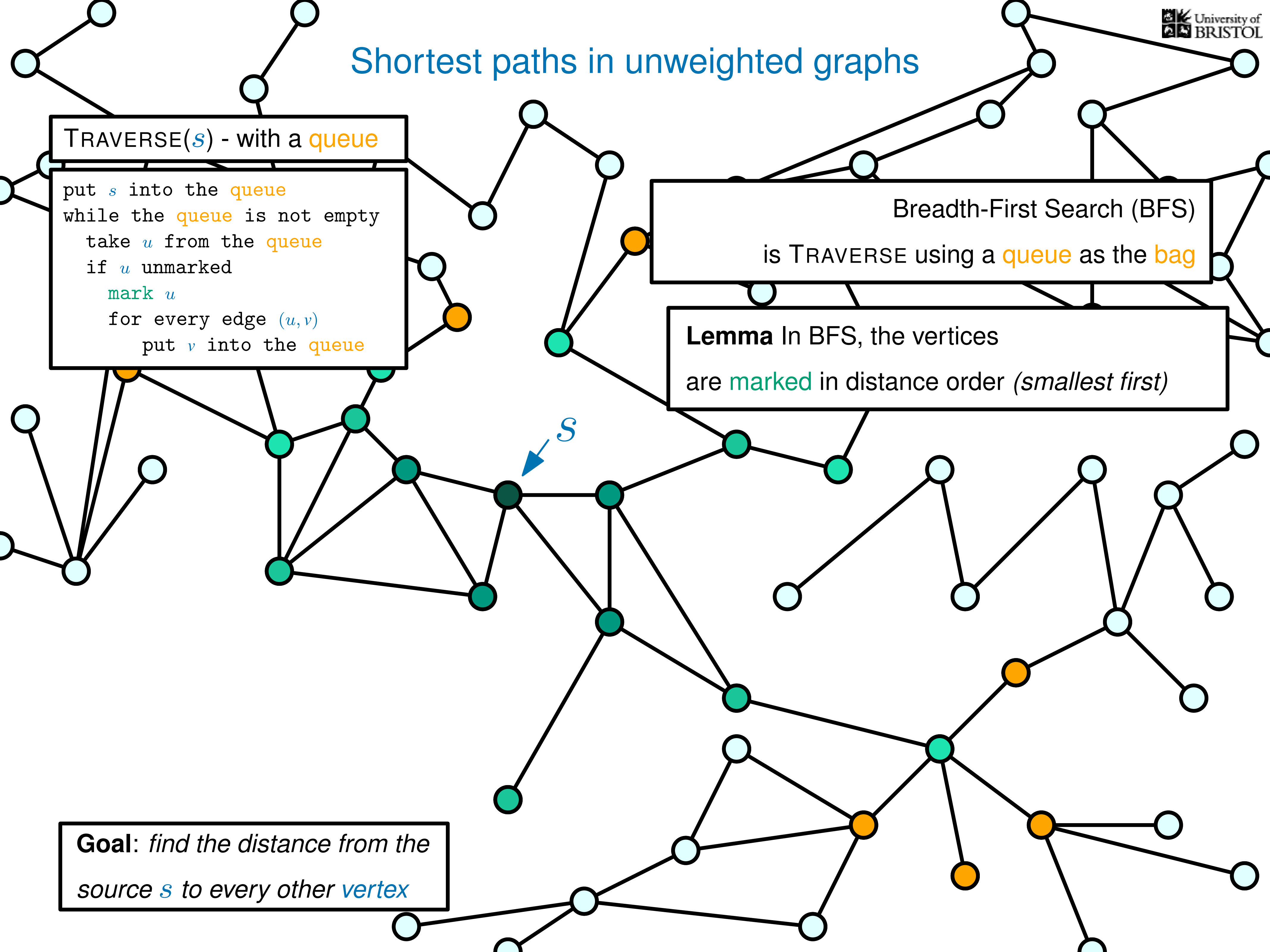
TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
      put v into the queue
```

Breadth-First Search (BFS)

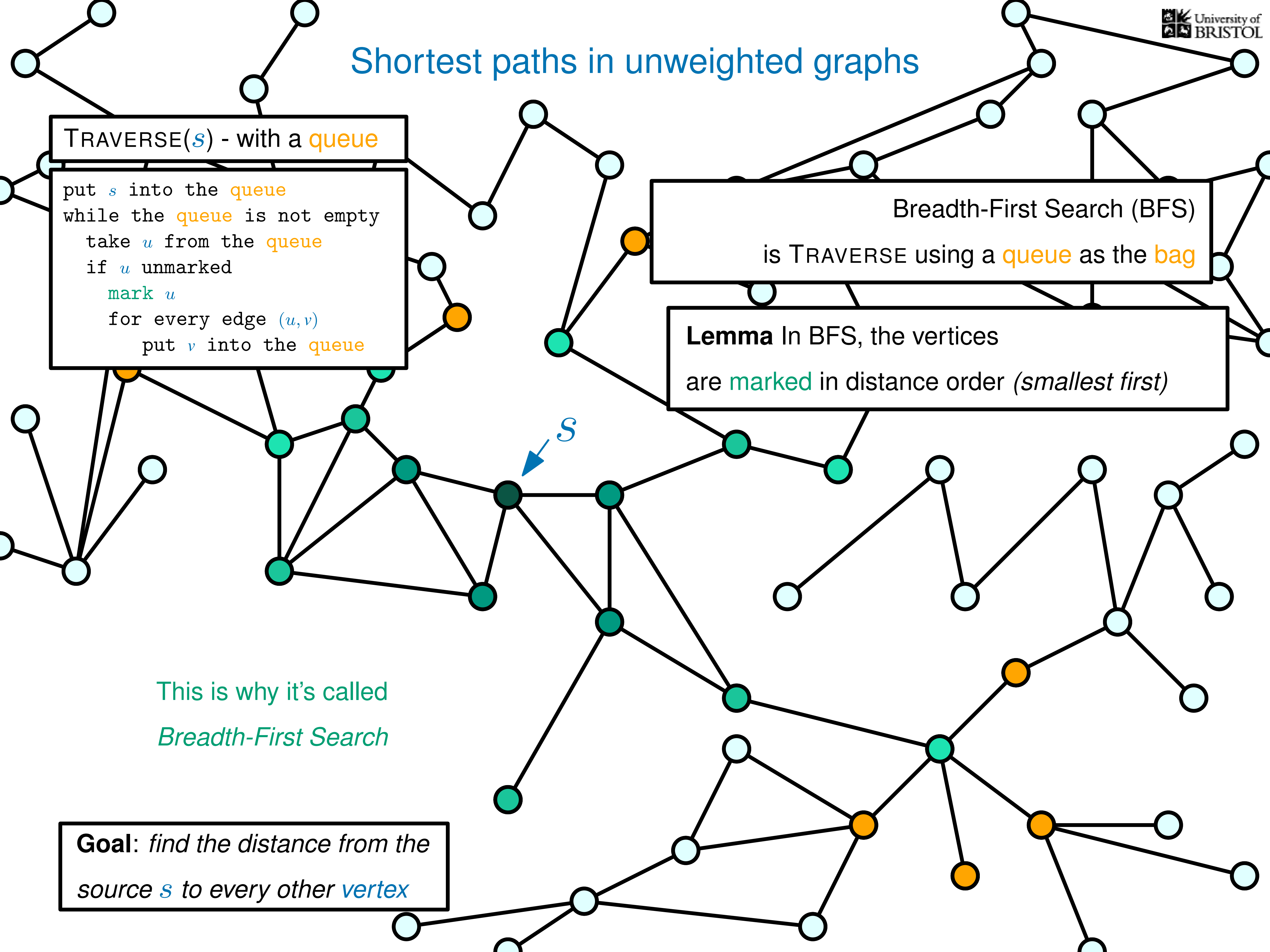
is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*



Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

This is why it's called *Breadth-First Search*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Base Case: BFS marks all vertices at distance 0 from s before marking any vertex at distance > 0 .

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Base Case: BFS marks all vertices at distance 0 from s before marking any vertex at distance > 0 .

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Base Case: BFS marks all vertices at distance 0 from s before marking any vertex at distance > 0 .

Inductive Hypothesis: Assume that BFS marks all vertices at distance $(i-1)$ before marking any vertex at distance $> (i-1)$.

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Base Case: BFS marks all vertices at distance 0 from s before marking any vertex at distance > 0 .

Inductive Hypothesis: Assume that BFS marks all vertices at distance $(i-1)$ before marking any vertex at distance $> (i-1)$.

When BFS marks a vertex at distance $(i-1)$,

it puts all its neighbours in the queue.

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Base Case: BFS marks all vertices at distance 0 from s before marking any vertex at distance > 0 .

Inductive Hypothesis: Assume that BFS marks all vertices at distance $(i-1)$ before marking any vertex at distance $> (i-1)$.

v is u's neighbour iff $(u, v) \in E$

When BFS marks a vertex at distance $(i-1)$,

it puts all its neighbours in the queue.

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Base Case: BFS marks all vertices at distance 0 from s before marking any vertex at distance > 0 .

Inductive Hypothesis: Assume that BFS marks all vertices at distance $(i-1)$ before marking any vertex at distance $> (i-1)$.

v is u's neighbour iff $(u, v) \in E$

When BFS marks a vertex at distance $(i-1)$,

it puts all its neighbours in the queue.

Immediately after marking *all* vertices at distance $(i-1)$:

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Base Case: BFS marks all vertices at distance 0 from s before marking any vertex at distance > 0 .

Inductive Hypothesis: Assume that BFS marks all vertices at distance $(i-1)$ before marking any vertex at distance $> (i-1)$.

v is u's neighbour iff $(u, v) \in E$

When BFS marks a vertex at distance $(i-1)$,

it puts all its neighbours in the queue.

Immediately after marking *all* vertices at distance $(i-1)$:

every vertex at distance i is in the queue.

no vertex at distance $> i$ has been put in the queue.

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Base Case: BFS marks all vertices at distance 0 from s before marking any vertex at distance > 0 .

Inductive Hypothesis: Assume that BFS marks all vertices at distance $(i-1)$ before marking any vertex at distance $> (i-1)$.

v is u's neighbour iff $(u,v) \in E$

When BFS marks a vertex at distance $(i-1)$,

it puts all its neighbours in the queue.

Immediately after marking *all* vertices at distance $(i-1)$:

every vertex at distance i is in the queue.

no vertex at distance $> i$ has been put in the queue.

because we haven't marked any of its neighbours

(they are all at distance $> (i-1)$)

Shortest paths in unweighted graphs

Traverse(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Lemma In BFS, the vertices are marked in distance order (smallest first)

Proof by induction (on the distance from s)

Base Case: BFS marks all vertices at distance 0 from s before marking any vertex at distance > 0 .

Inductive Hypothesis: Assume that BFS marks all vertices at distance $(i-1)$ before marking any vertex at distance $> (i-1)$.

v is u's neighbour iff $(u, v) \in E$

When BFS marks a vertex at distance $(i-1)$,

it puts all its neighbours in the queue.

Immediately after marking *all* vertices at distance $(i-1)$:

every vertex at distance i is in the queue.

no vertex at distance $> i$ has been put in the queue.

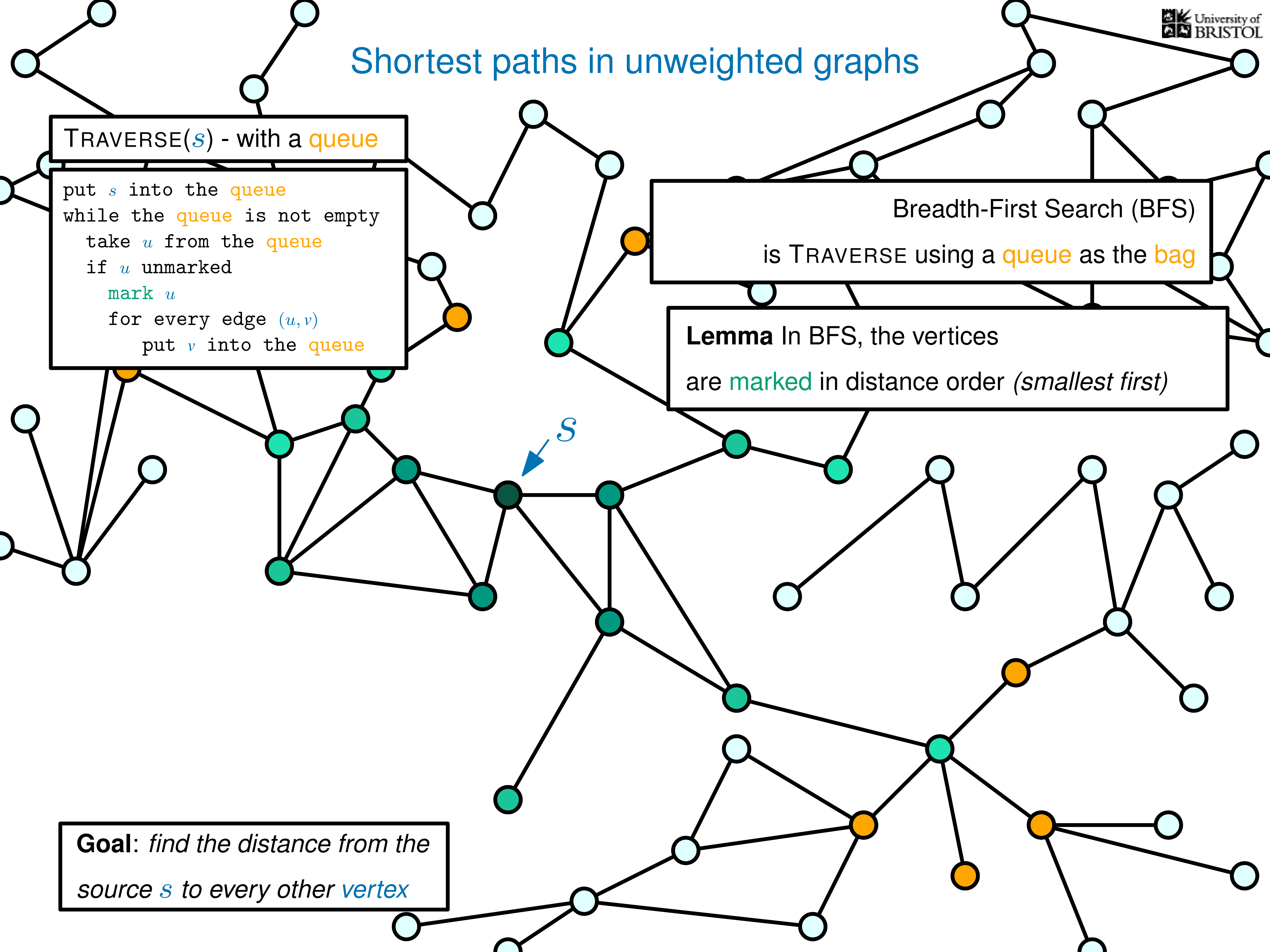
because we haven't marked any of its neighbours

(they are all at distance $> (i-1)$)

Therefore, *all* vertices at distance i will be marked before *any* vertex at distance $> i$

University of BRISTOL

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

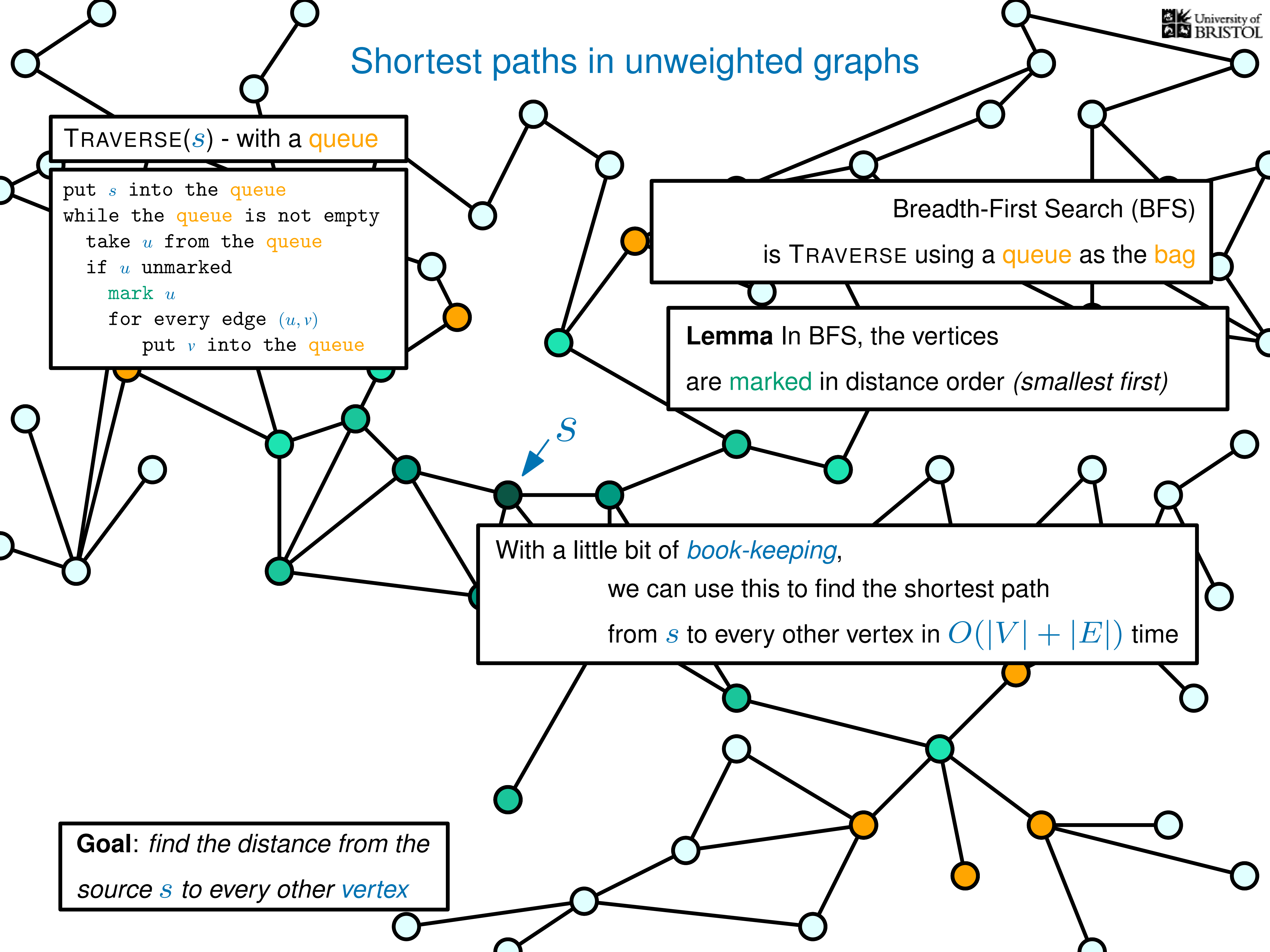
Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs



TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

With a little bit of *book-keeping*, we can use this to find the shortest path from s to every other vertex in $O(|V| + |E|)$ time

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices are marked in distance order *(smallest first)*

s

With a little bit of *book-keeping*, we can use this to find the shortest path from s to every other vertex in $O(|V| + |E|)$ time

This also works for **directed**, unweighted graphs

Goal: *find the distance from the source s to every other vertex*

Shortest paths in unweighted graphs

TRAVERSE(s) - with a queue

```
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
```

Breadth-First Search (BFS)

is TRAVERSE using a queue as the bag

Lemma In BFS, the vertices

are marked in distance order *(smallest first)*

s

With a little bit of *book-keeping*,

we can use this to find the shortest path

from s to every other vertex in $O(|V| + |E|)$ time

This also works for **directed**, unweighted graphs

It does not work for weighted graphs

Goal: *find the distance from the source s to every other vertex*

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\mathtt{dist}(v)$ gives the distance

between s and v

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

How does this affect the time complexity?

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance

between s and v

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

How does this affect the time complexity?

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\mathtt{dist}(v)$ gives the distance

between s and v

Time complexity:

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

How does this affect the time complexity?

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

Time complexity:

$O(|V|)$ time to initialise an array dist

dist(v) gives the distance

between s and v

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

How does this affect the time complexity?

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
        put v into the queue
        if dist(v) = ∞ (NEW!)
          dist(v) = dist(u) + 1 (NEW!)
```

Time complexity:

$O(|V|)$ time to initialise an array dist

$O(1)$ time to set the distance to s

$\texttt{dist}(v)$ gives the distance between s and v

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

How does this affect the time complexity?

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

Time complexity:

$O(|V|)$ time to initialise an array dist

$O(1)$ time to set the distance to s

$O(1)$ time whenever we put into the queue

(so this doesn't change the complexity)

dist(v) gives the distance

between s and v

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

How does this affect the time complexity?

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance

between s and v

Time complexity:

$O(|V|)$ time to initialise an array dist

$O(1)$ time to set the distance to s

$O(1)$ time whenever we put into the queue

(so this doesn't change the complexity)

so the overall complexity is $O(|V| + |E|)$

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance

between s and v

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE to track the distances from s

Why does this work?

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance between s and v

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

dist(v) gives the distance

between s and v

Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

dist(v) gives the distance

between s and v

Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

dist(v) gives the distance

between s and v

Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, dist(v) is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\mathtt{dist}(v)$ gives the distance

between s and v

Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\mathtt{dist}(v)$ is set when it is first 'discovered'

and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

University of BRISTOL

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\mathtt{dist}(v)$ gives the distance

between s and v

Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\mathtt{dist}(v)$ is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

BFS sets $\mathtt{dist}(v) = \mathtt{dist}(u) + 1$

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance

between s and v

Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\texttt{dist}(v)$ is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

BFS sets $\texttt{dist}(v) = \texttt{dist}(u) + 1$

Assume for a contradiction that there is a path

from s to v with length $< \texttt{dist}(u) + 1$

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

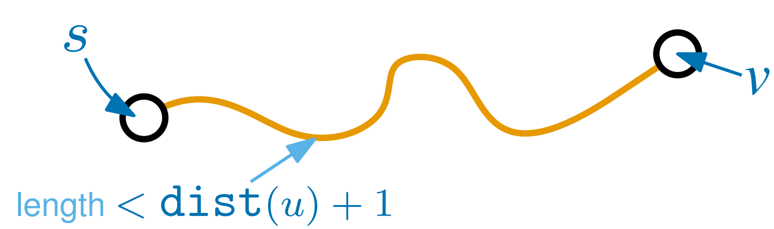
to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
   take u from the queue
   if u unmarked
     mark u
     for every edge (u, v)
        put v into the queue
        if dist(v) = ∞ (NEW!)
          dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance

between s and v



Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\texttt{dist}(v)$ is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

BFS sets $\texttt{dist}(v) = \texttt{dist}(u) + 1$

Assume for a contradiction that there is a path

from s to v with length $< \texttt{dist}(u) + 1$

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

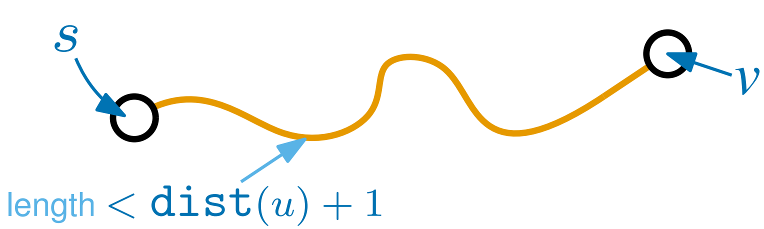
to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
   take u from the queue
   if u unmarked
     mark u
     for every edge (u, v)
        put v into the queue
        if dist(v) = ∞ (NEW!)
          dist(v) = dist(u) + 1  (NEW!)
```

$\texttt{dist}(v)$ gives the distance

between s and v



Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\texttt{dist}(v)$ is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

BFS sets $\texttt{dist}(v) = \texttt{dist}(u) + 1$

Assume for a contradiction that there is a path

from s to v with length $< \texttt{dist}(u) + 1$

Let x be the previous vertex on this path

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

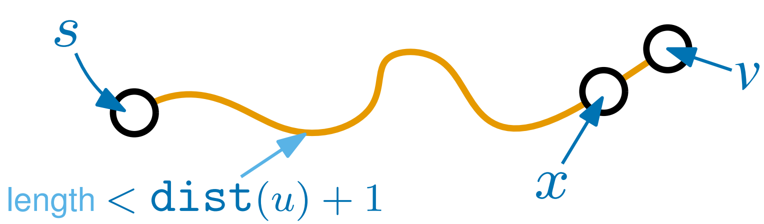
to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u, v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance

between s and v



Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\texttt{dist}(v)$ is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

BFS sets $\texttt{dist}(v) = \texttt{dist}(u) + 1$

Assume for a contradiction that there is a path

from s to v with length $< \texttt{dist}(u) + 1$

Let x be the previous vertex on this path

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

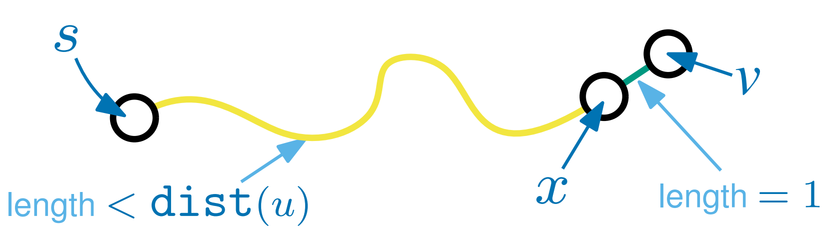
to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
   take u from the queue
   if u unmarked
     mark u
     for every edge (u, v)
         put v into the queue
         if dist(v) = ∞ (NEW!)
           dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance

between s and v



Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\texttt{dist}(v)$ is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

BFS sets $\texttt{dist}(v) = \texttt{dist}(u) + 1$

Assume for a contradiction that there is a path

from s to v with length $< \texttt{dist}(u) + 1$

Let x be the previous vertex on this path

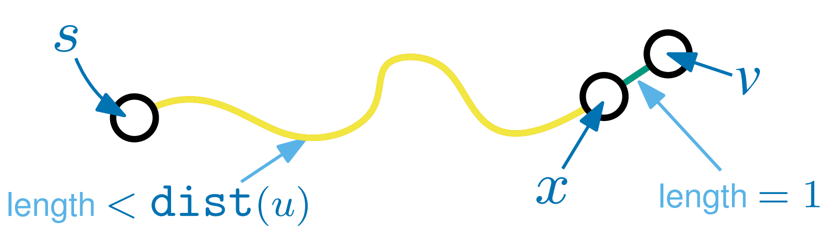
Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\mathtt{dist}(v)$ gives the distance between s and v



Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\mathtt{dist}(v)$ is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

BFS sets $\mathtt{dist}(v) = \mathtt{dist}(u) + 1$

Assume for a contradiction that there is a path from s to v with length $< \mathtt{dist}(u) + 1$

Let x be the previous vertex on this path

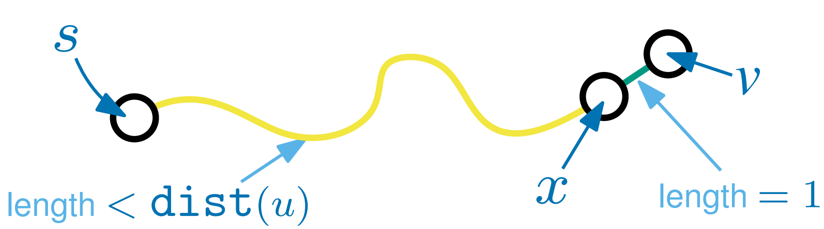
Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
   take u from the queue
   if u unmarked
      mark u
      for every edge (u, v)
         put v into the queue
         if dist(v) = ∞ (NEW!)
            dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance between s and v



Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\texttt{dist}(v)$ is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

BFS sets $\texttt{dist}(v) = \texttt{dist}(u) + 1$

Assume for a contradiction that there is a path

from s to v with length $< \texttt{dist}(u) + 1$

Let x be the previous vertex on this path

x has distance $< \texttt{dist}(u)$ so was marked before u

(by the **Lemma**) and 'discovered' v first

Shortest Paths using BFS

We've added four (NEW!) lines to TRAVERSE

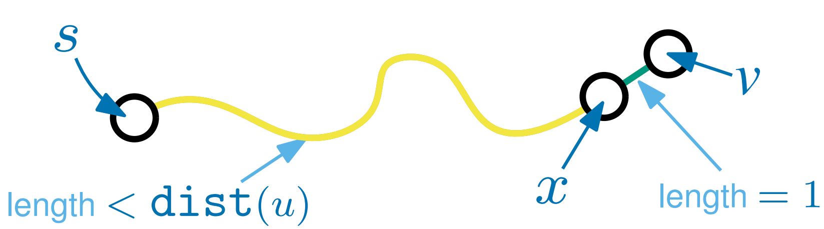
to track the distances from s

BFS(s)

```
for all v, set dist(v) = ∞ (NEW!)
set dist(s) = 0 (NEW!)
put s into the queue
while the queue is not empty
  take u from the queue
  if u unmarked
    mark u
    for every edge (u,v)
       put v into the queue
       if dist(v) = ∞ (NEW!)
         dist(v) = dist(u) + 1 (NEW!)
```

$\texttt{dist}(v)$ gives the distance

between s and v



Why does this work?

Lemma In BFS, the vertices are marked in distance order (smallest first)

Correctness sketch: (using the Lemma)

For each v, $\texttt{dist}(v)$ is set when it is first 'discovered' and inserted into the queue. (and it never changes)

Let u be the vertex which first 'discovered' v

BFS sets $\texttt{dist}(v) = \texttt{dist}(u) + 1$

Assume for a contradiction that there is a path

from s to v with length $< \texttt{dist}(u) + 1$

Let x be the previous vertex on this path

x has distance $< \texttt{dist}(u)$ so was marked before u

(by the **Lemma**) and 'discovered' v first

Contradiction!

V is the set of vertices
$|V|$ *is the number of vertices*

Conclusion

E is the set of edges
$|E|$ *is the number of edges*

TRAVERSE visits every vertex in a connected graph in $O(|E|)$ time

(with an $O(1)$ time bag)

- the traversal order depends on the type of bag

(and it works for directed graphs too)

with a Queue the algorithm is called

Breadth First Search (BFS)

Applications

Max-Flow

Testing whether a graph is bipartite

Shortest paths in unweighted graphs

take $O(|V| + |E|)$ time using BFS

(works for directed graphs too)

with a Stack the algorithm is called

Depth First Search (DFS)

Applications

Finding (strongly) connected components

Topicologically sorting a Directed Acyclic Graph

Testing for planarity

Question

What does TRAVERSE do on an unconnected graph?