Graph Traversal

❖ Graph Traversal

Many of the above problems can be solved by

对边

- systematic exploration of a graph, via the edges

Algorithms for this typically require us to remember

- what vertices we have already visited
- the path we followed while visiting them

Since many graph search algorithms are recursive

- above information needs to be stored globally
- and updated by individual calls to the recursive function

Systematic exploration like this is called traversal or search.

遍历

❖ ... Graph Traversal

Consider two related problems on graphs ...

源头 目的地

- is there a path between two given vertices (*src*,*dest*)?
- what is the sequence of vertices from *src* to *dest*?

An approach to solving this problem:

- examine vertices adjacent to *src*
- if any of them is *dest*, then done
- otherwise try vertices two edges from *src*
- repeat looking further and further from *src*

The above summarises one form of graph traversal.

❖ ... Graph Traversal

There are two strategies for graph traversal/search ...

Depth-first search (DFS)

- favours following path rather than neighbours
- can be implemented recursively or iteratively (via stack)
- full traversal produces a depth-first spanning tree

Breadth-first search (BFS)

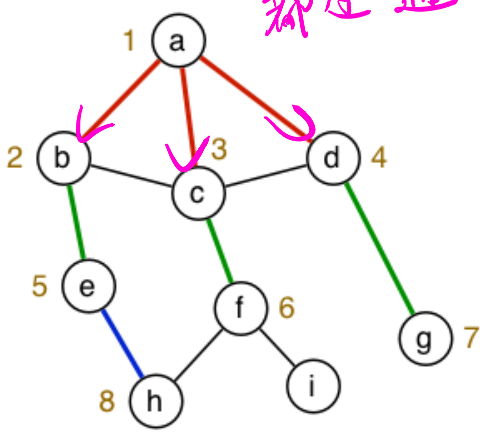
- favours neighbours rather than path following
- can be implemented iteratively (via queue)
- full traversal produces a breadth-first spanning tree

The method on the previous slide is effectively breadth-first traversal.

❖ ... Graph Traversal

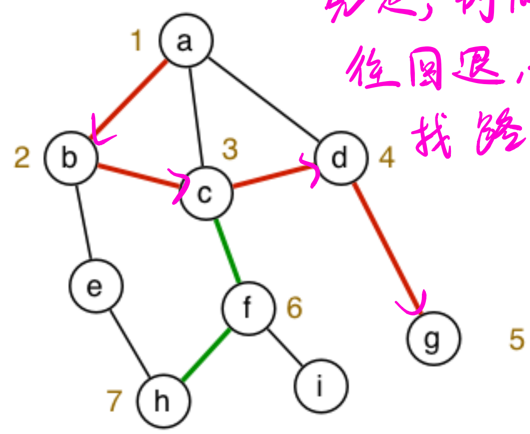
Comparison of BFS/DFS search for checking **`hasPath(a,h)`**?

都走一遍



Breadth-first Search

先走，到底再往回退，再找路



Depth-first Search

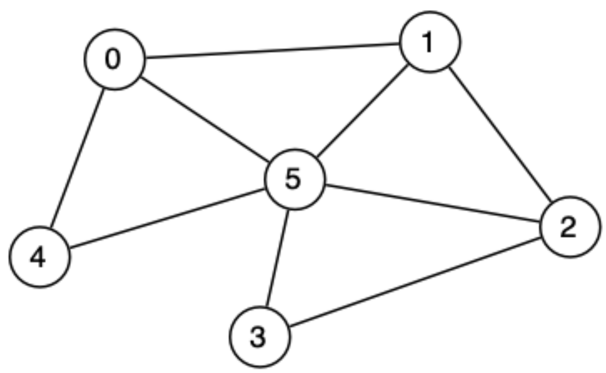
Both approaches ignore some edges by remembering previously visited vertices.

❖ ... Graph Traversal

A spanning tree of a graph

- includes all vertices, using a subset of edges, without cycles

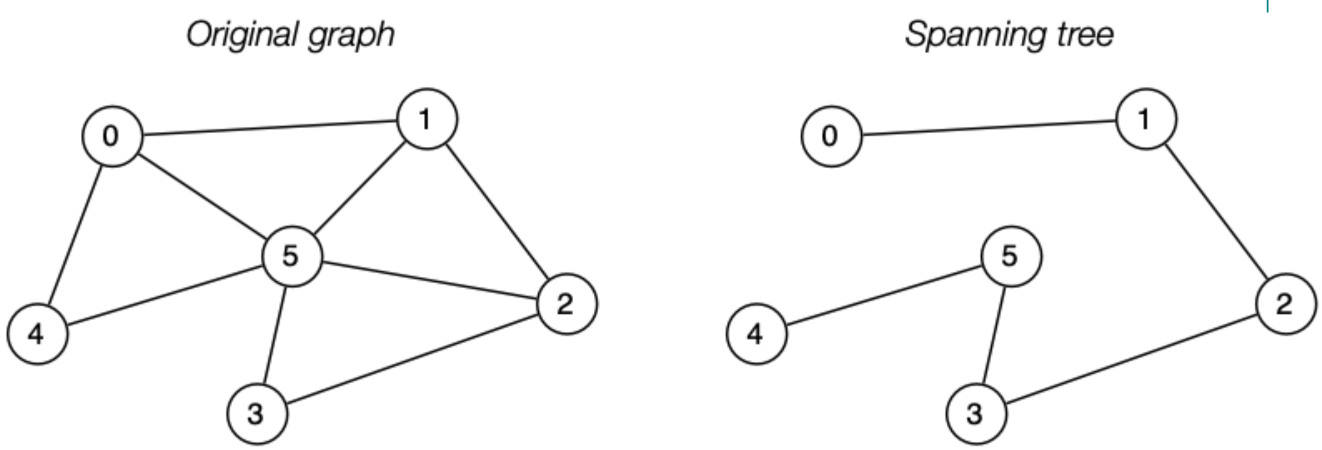
Consider the following graph:



Consider how DFS and BFS could produce its spanning tree

❖ ... Graph Traversal

Spanning tree resulting from DFS ...



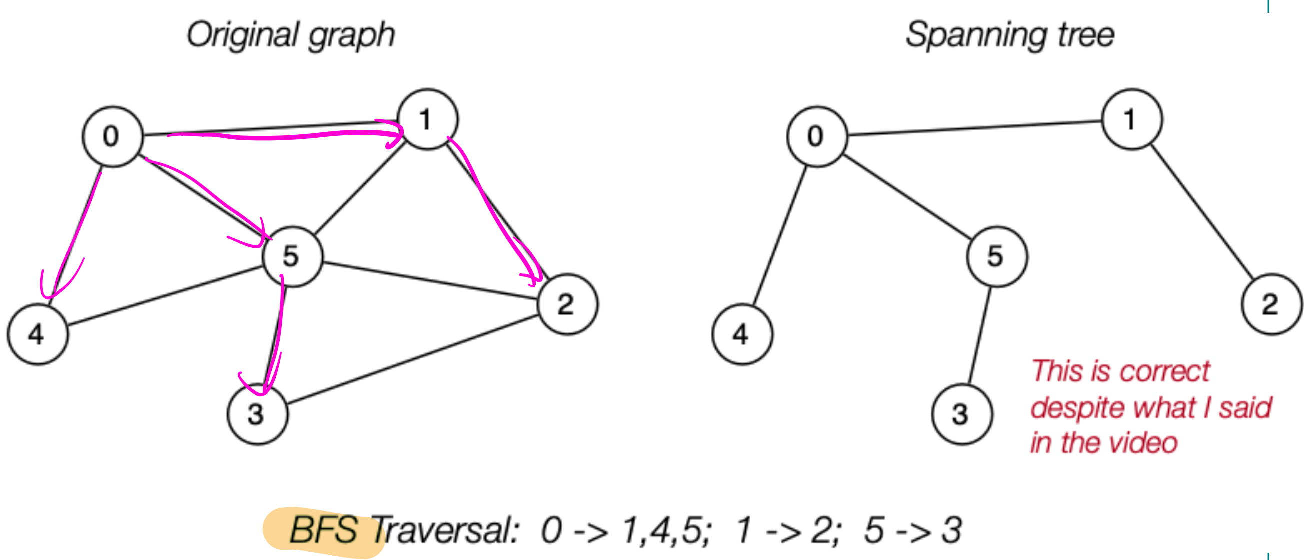
DFS Traversal: 0 -> 1 -> 2 -> 3 -> 5 -> 4

从小到大

Note: choose neighbours in ascending order

❖ ... Graph Traversal

Spanning tree resulting from BFS ...



Note: choose neighbours in ascending order

❖ Depth-first Search

Depth-first search can be described recursively as

```
visited = {}          array

depthFirst(G,v):   v first
|  visited = visited ∪ {v}    Graph
|  for all (v,w) ∈ edges(G) do
|  |  if w ∉ visited then
|  |  |  depthFirst(G,w)
|  |  end if
|  end for
```

w is a neighbour of v.

The recursion induces backtracking

❖ ... Depth-first Search

Recursive DFS path checking

```
visited = {}

hasPath(G,src,dest):
|   Input  graph G, vertices src,dest
|   Output true if there is a path from src to dest,
|          false otherwise
|
|   return dfsPathCheck(G,src,dest)
```

Requires wrapper around recursive function
`dfsPathCheck()`

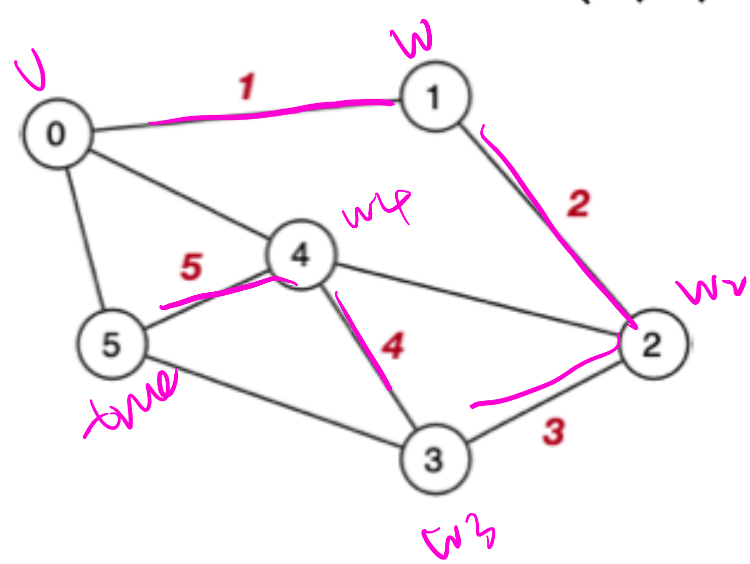
❖ ... Depth-first Search

Recursive function for path checking

v to dest

```
dfsPathCheck(G,v,dest):
|  visited = visited ∪ {v}                      v first
|  for all (v,w) ∈ edges(G) do
|  |  if w=dest then     // found edge to dest
|  |     return true
|  |  else if w ∉ visited then
|  |     if dfsPathCheck(G,w,dest) then
|  |        return true   // found path via w to dest
|  |     end if
|  |  end if
|  end for
|  return false   // no path from v to dest
```

Tracing the execution of **`dfsPathCheck(G,0,5)`** on:



Reminder: we consider neighbours in ascending order

Clearly does not find the shortest path

❖ DFS Cost Analysis

Cost analysis:

- each vertex visited at most once ⇒ cost = *O(V)*
- visit all edges incident on visited vertices ⇒ cost = *O(E)*
    - assuming an adjacency list representation

Time complexity of DFS: *O(V+E)* (adjacency list representation)

❖ Path Finding

Knowing whether a path exists can be useful

Knowing what the path is, is even more useful

Strategy:

- record the previously visited node as we search
- so that we can then trace path (backwards) through graph

Requires a global array (not a set):

- `visited[v]` contains vertex `w` from which we reached `v`

❖ ... Path Finding

Function to find path src→dest and print it

```
visited[] // store previously visited node
          // for each vertex 0..nV-1

findPath(G,src,dest):
|  Input graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  visited[src]=src   // starting node of the path
|  if dfsPathCheck(G,src,dest) then
|  |  // show path in dest..src order
|  |  v=dest
|  |  while v≠src do
|  |     print v"-"
|  |     v=visited[v]
|  |  end while
|  |  print src
|  end if
```

print the path

❖ ... Path Finding

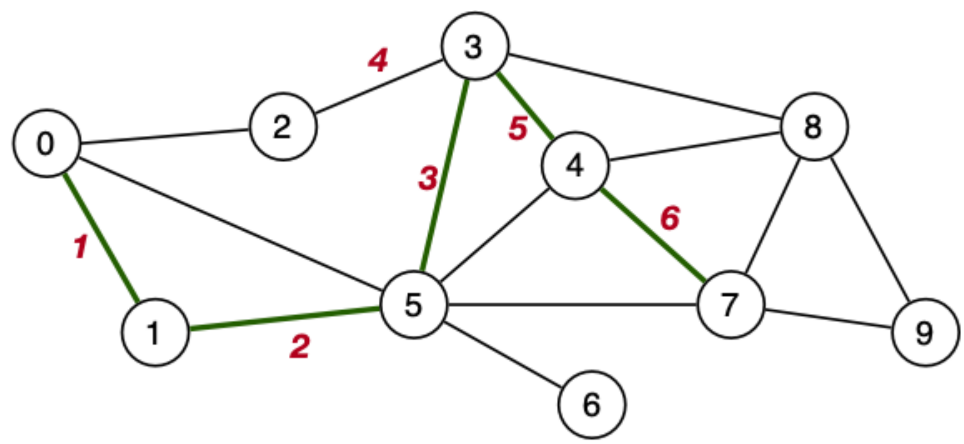
Recursive function to build path in **`visited[]`**

```
dfsPathCheck(G,v,dest):
|  for all (v,w) ∈ edges(G) do
|  |  if visited[w] = -1 then
|  |     visited[w] = v
|  |     if w = dest then   // found edge from v to dest
|  |        return true
|  |     else if dfsPathCheck(G,w,dest) then
|  |        return true   // found path via w to dest
|  |     end if
|  |  end if
|  end for
|  return false   // no path from v to dest
```

❖ ... Path Finding

The **`visited[]`** array after **`dfsPathCheck(G,0,7)`** succeeds

visited[0] = 0



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
visited	0	0	3	5	3	1	5	4	-1	-1

prev of 5 is 1

7 - 4 - 3 - 5 - 1 - 0

$\longrightarrow$

v = visited[v]

v = visited[7] = 4

[4] = 3

[3] = 5

⋮

❖ ... Path Finding

DFS can also be described non-recursively (via a stack):

```
visited[] // store previously visited node
          // for each vertex 0..nV-1

findPathDFS(G,src,dest):
|  Input graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  found=false
|  visited[src]=src
|  push src onto new stack S
|  while not found ∧ S is not empty do
|  |  pop v from S
|  |  if v=dest then
|  |     found=true
|  |  else
|  |  |  for each (v,w)∈edges(G) with visited[w]=-1 do
|  |  |     visited[w]=v
|  |  |     push w onto S
|  |  |  end for
|  |  end if
|  end while
|  if found then
|     display path in dest..src order
|  end if
```

一开始是false 0 0

Uses standard stack operations ... Time complexity is still *O(V+E)*

❖ Breadth-first Search

Basic approach to breadth-first search (BFS):

- visit and mark current vertex
- visit all neighbours of current vertex
- then consider neighbours of neighbours

Notes:

- tricky to describe recursively
- but a minor variation on non-recursive DFS search works
  $\Rightarrow$ switch the *stack* for a *queue*

❖ ... Breadth-first Search

BFS path finding algorithm:

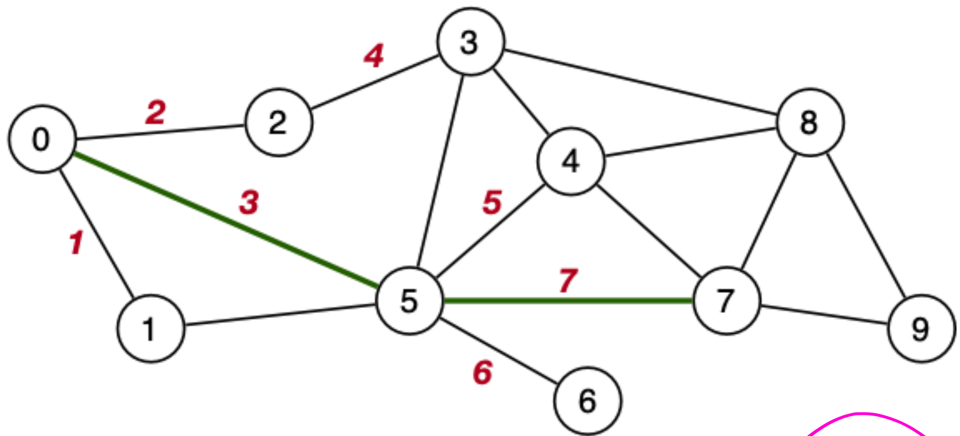
```
visited[] // store previously visited node
          // for each vertex 0..nV-1

findPathBFS(G,src,dest):
|  Input  graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  found=false
|  visited[src]=src
|  enqueue src into queue Q
|  while not found ∧ Q is not empty do
|  |  dequeue v from Q
|  |  if v=dest then
|  |     found=true
|  |  else                                          neighbour
|  |  |  for each (v,w) ∈ edges(G) with visited[w]=-1 do
|  |  |     visited[w]=v
|  |  |     enqueue w into Q
|  |  |  end for
|  |  end if
|  end while
|  if found then
|     display path in dest..src order
|  end if
```

Uses standard queue operations (enqueue, dequeue, check if empty)

❖ ... Breadth-first Search

The **`visited[]`** array after **`findPathBFS(G,0,7)`** succeeds



visited

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	0	0	2	5	0	5	5	-1	-1

never go

初始 -1

❖ ... Breadth-first Search

Time complexity of BFS: *O(V+E)* (same as DFS)

BFS finds a "shortest" path

- based on minimum # edges between *src* and *dest*.
- stops with first-found path, if there are multiple ones

In many applications, edges are weighted and we want path

- based on minimum sum-of-weights along path *src* .. *dest*

We discuss weighted/directed graphs later.