CSE 12:

Basic data structures and object-oriented design

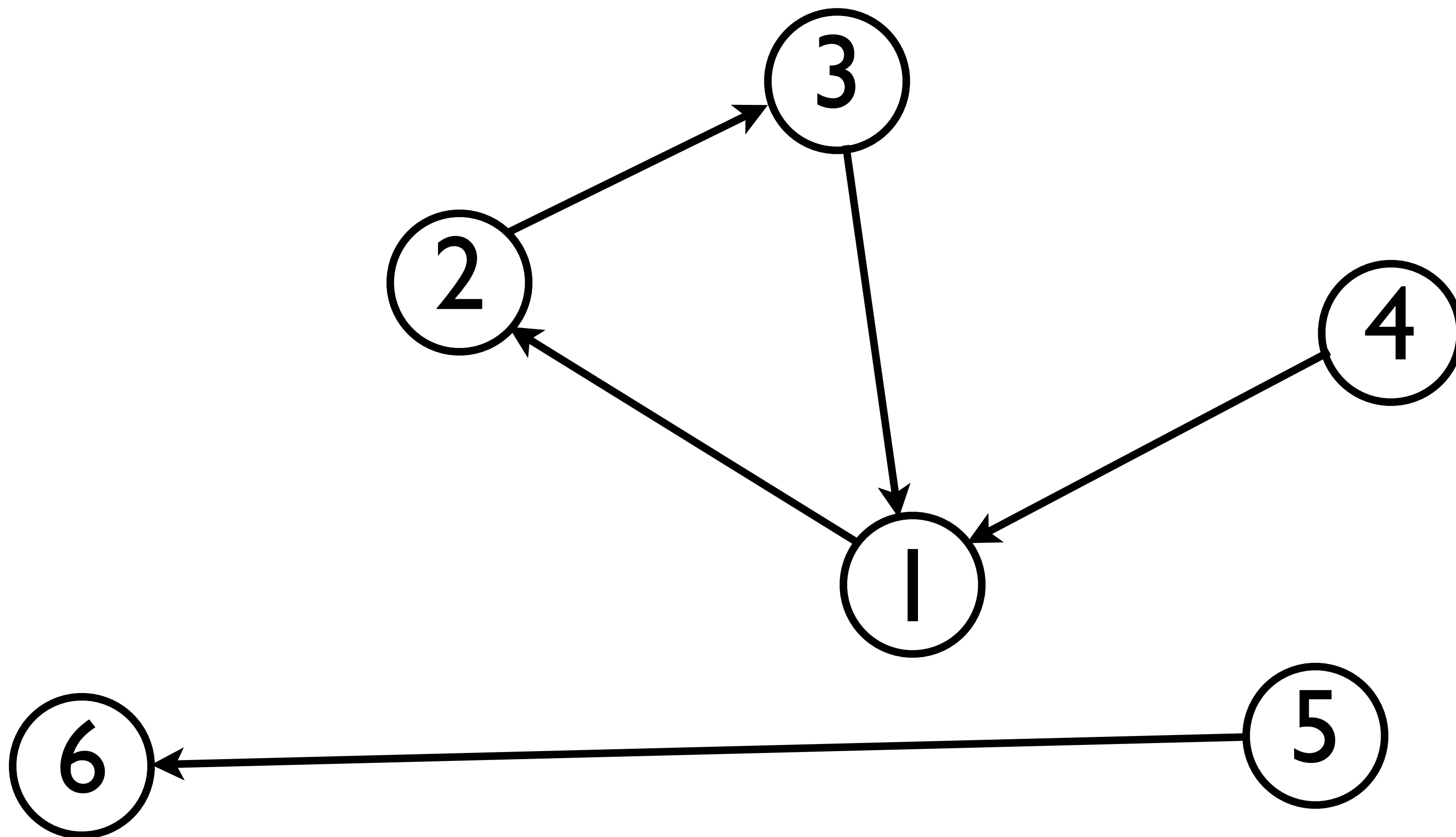
Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Fifteen
31 July 2012

Graphs, continued

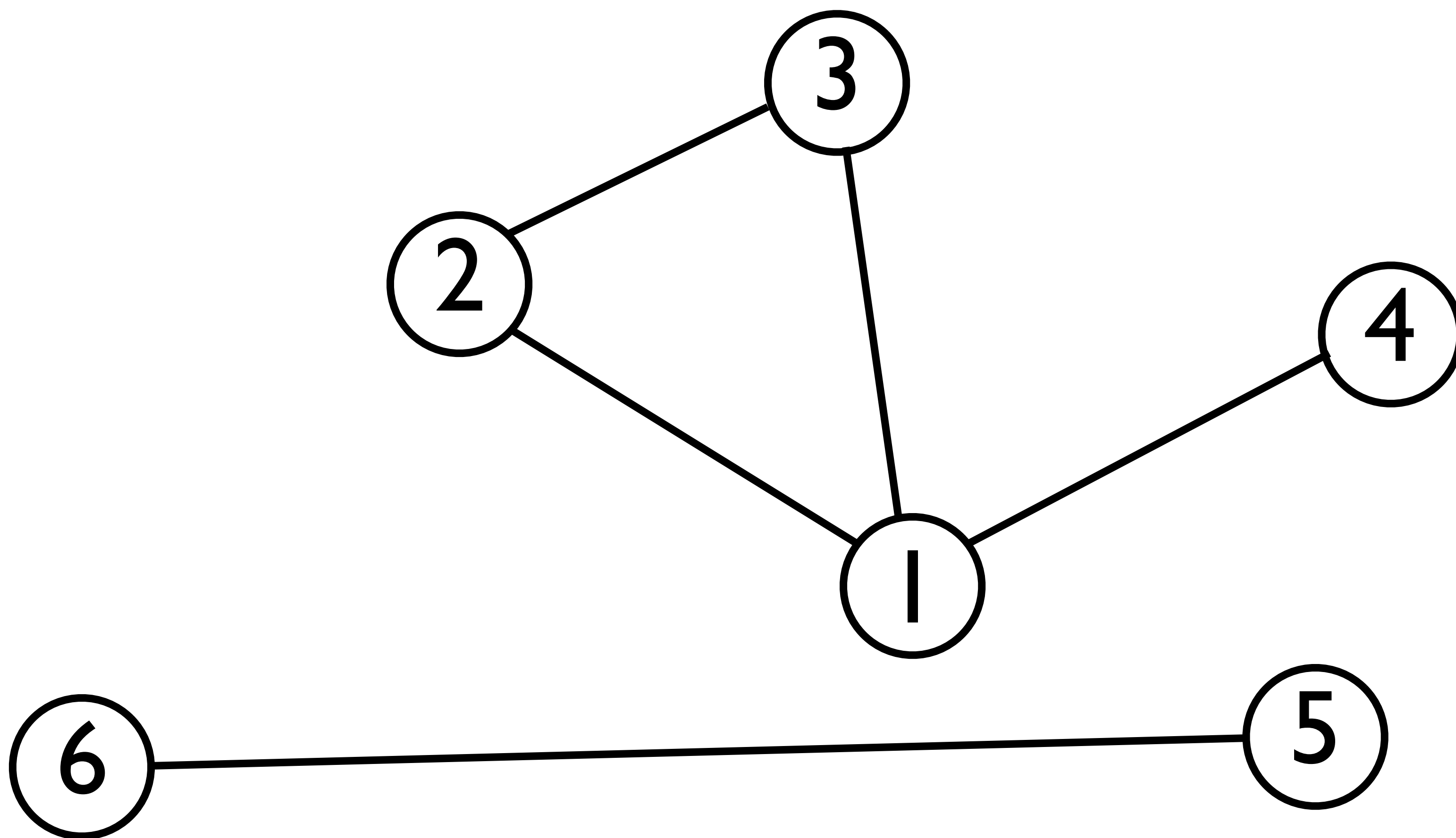
Graphs

- In the graph below, N = { 1, 2, 3, 4, 5, 6 }.
- An *edge* in a directed graph from node m to node n can be described as an *ordered pair* (m, n).
- In the graph below, E = { (2, 3), (3, 1), (1, 2), (4, 1), (5, 6) }.



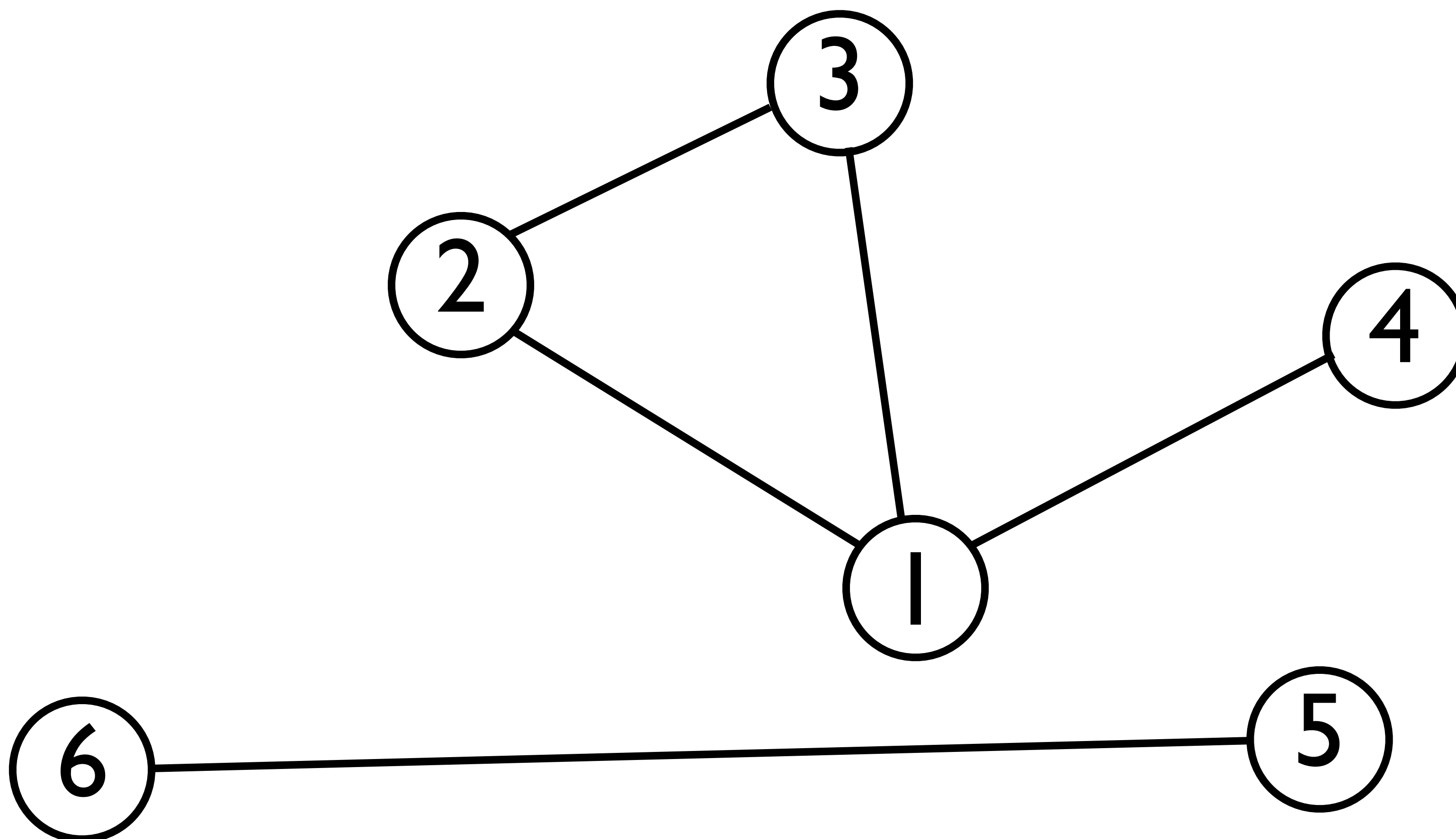
Graphs

- If a graph is undirected, then for every edge $(m, n) \in E$, we also have $(n, m) \in E$.
- For the graph below, $E = \{ (2, 3), (3, 2), (1, 3), (3, 1), (1, 2), (2, 1), (1, 4), (4, 1), (5, 6), (6, 5) \}$.



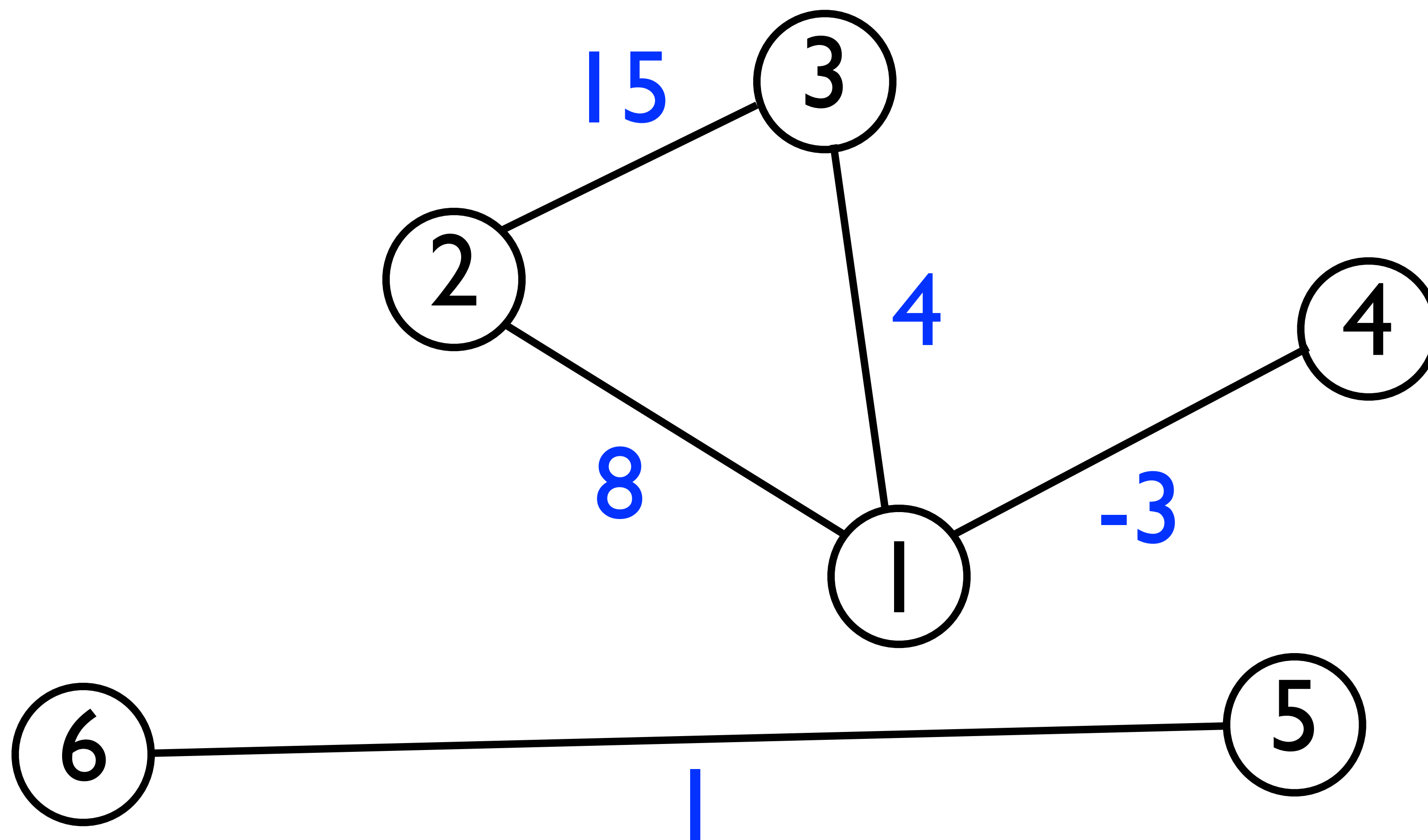
Graphs

- Whenever $(m, n) \in E$, we say that node m is **adjacent** (or **connected**) to node n.



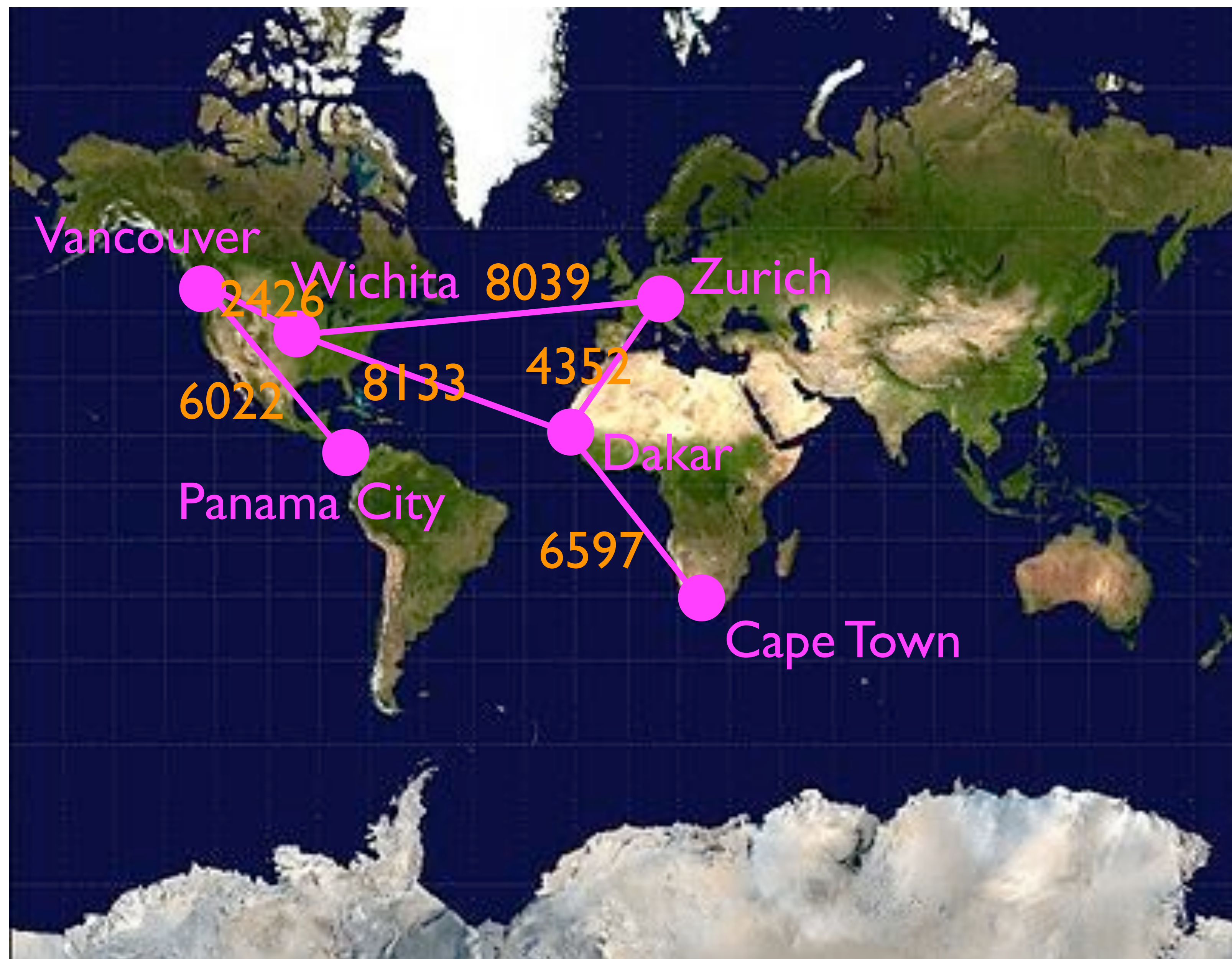
Graphs

- In some graphs, edges have **weights** associated with them to represent distance, cost, etc.
  - In this case, an edge can be represented as an ordered triplet (m, n, w_{mn}) where w_{mn} is the weight from m to n.



Graphs

- An example of a weighted graph is an airline map that shows *cities* connected by *flights*, and the weight of each edge is the *distance* (km) between those cities.



Representing graphs

- To use graphs as a data structure, we must devise a way of representing a graph in memory.
- Let *N* be the set of nodes and *E* be the set of edges.
- The number of nodes is $|N|$, and the number of edges is $|E|$.
- To represent the set of *nodes* in memory, we can use an $|N|$-element array, where each node is assigned a unique index.
  - This is both time- and space-efficient.

Representing graphs

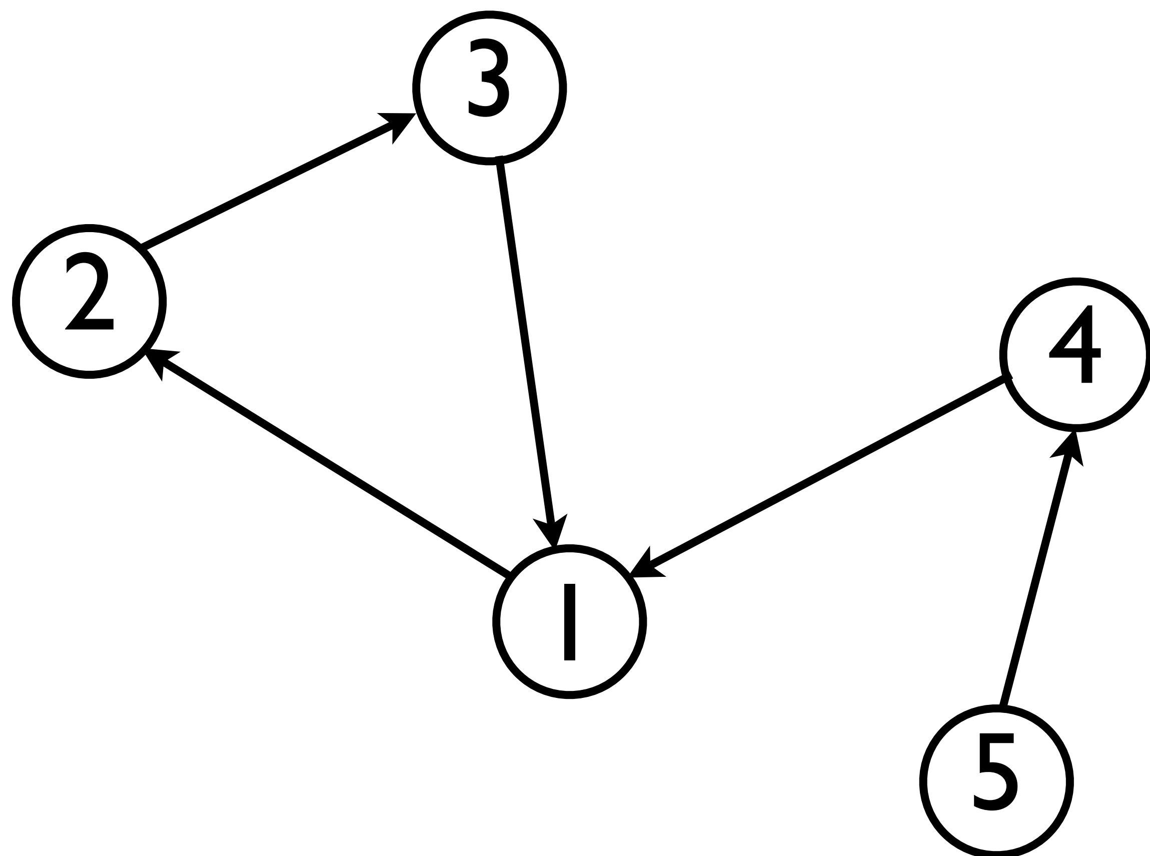
- To represent the set of edges, we can use two alternative representations:
  - An **adjacency matrix** A for the whole graph.
  - An **adjacency list** for every node $m \in N$.

Adjacency matrices

- An **adjacency matrix** *A* is an $|N|$ x $|N|$ matrix, where $|N|$ is the number of nodes in the graph.
  - For an *unweighted* graph, the (mn)th entry of A contains a 1 or a 0 depending on whether edge $(m, n) \in E$.
  - For a *weighted* graph, the (mn)th entry of A contains the *weight* of edge $(m, n) \in E$.
    - If $(m, n) \notin E$, then we can store either 0, infinity, or null (depending on what's most useful).

Adjacency matrices

Example for *directed* graph:



n (columns), *m* (rows):

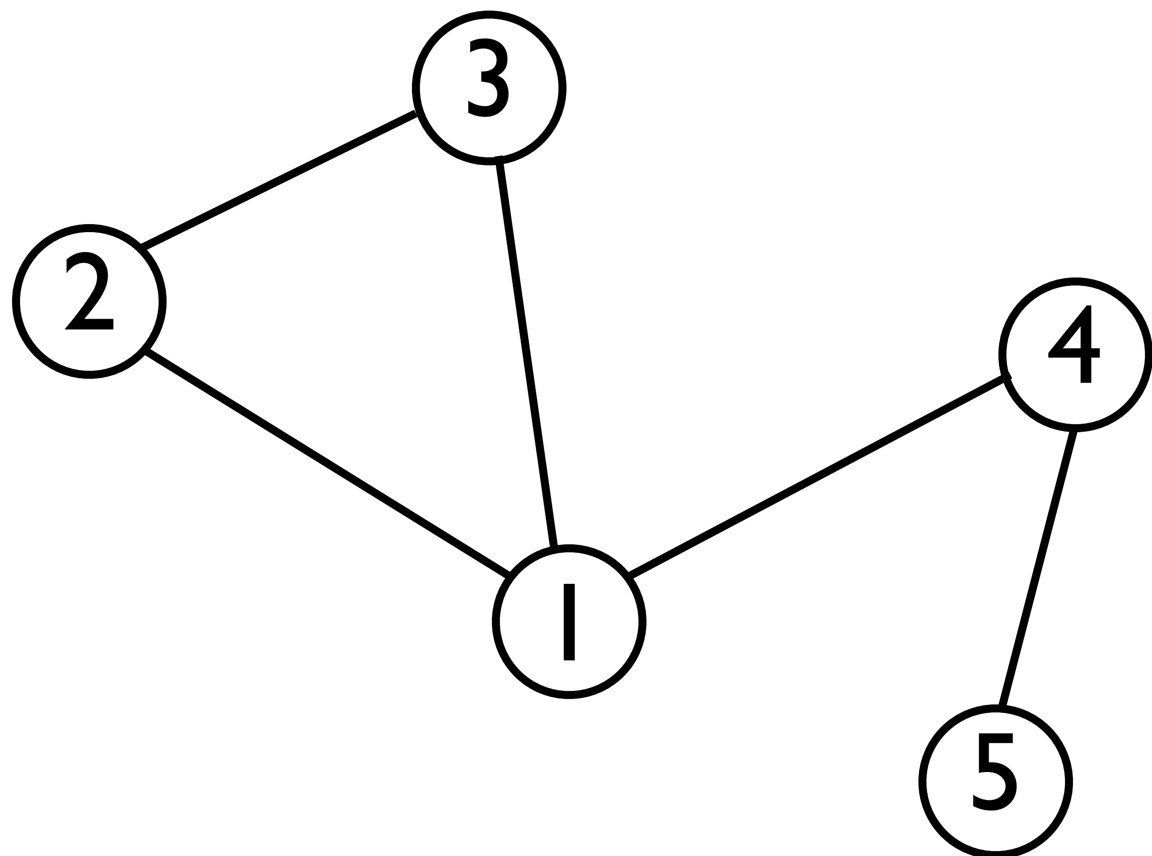
	1	2	3	4	5
1		1			
2			1		
3	1				
4	1				
5				1	

(All "blank" entries are 0.)

Adjacency matrices

Example for *undirected* graph:

In an *undirected* graph, the adjacency matrix A equals its own transpose (i.e., $A = A^T$).



n (columns), *m* (rows):

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

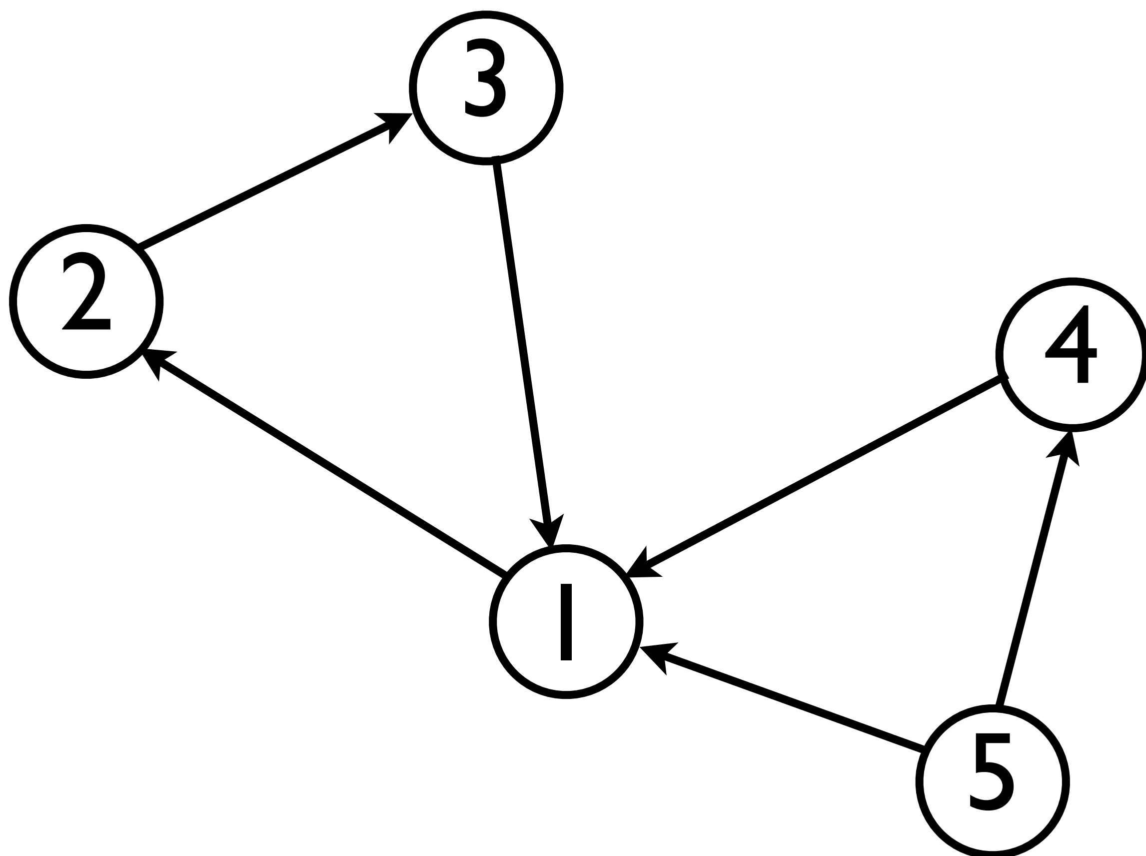
(All "blank" entries are 0.)

Adjacency matrices

- Adjacency matrices offer *fast access* to the presence/absence of any edge in the graph.
- However, for graphs in which edges are *sparse*, they are space-inefficient ($O(|N|^2)$).
- A space-saving (but slower) alternative is adjacency lists...

Adjacency lists

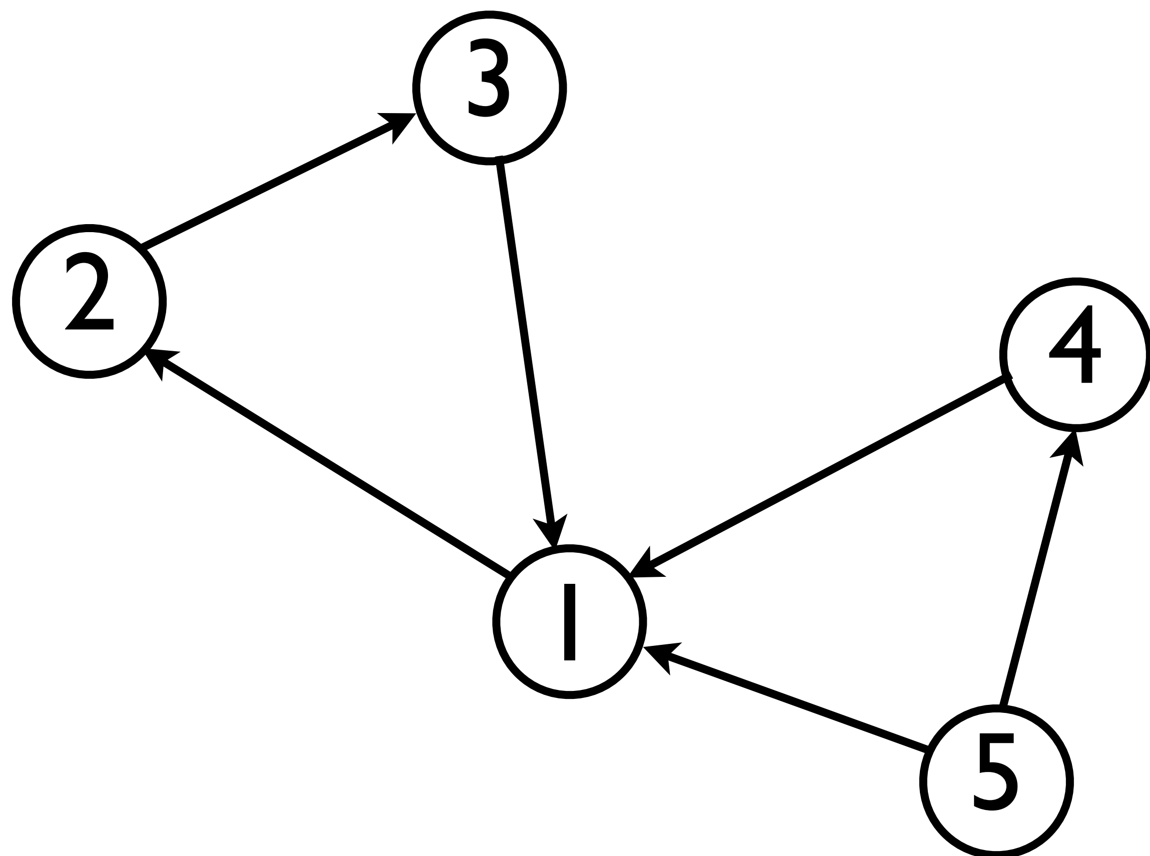
- With adjacency lists, every node maintains a *list* of other nodes to which it is connected.



Node 1: { 2 }
Node 2: { 3 }
Node 3: { 1 }
Node 4: { 1 }
Node 5: { 4, 1 }

Adjacency lists

- Adjacency lists require only $O(|E|)$ space to store all the edges.
- However, they require $O(|E|)$ time to *find* a particular edge.



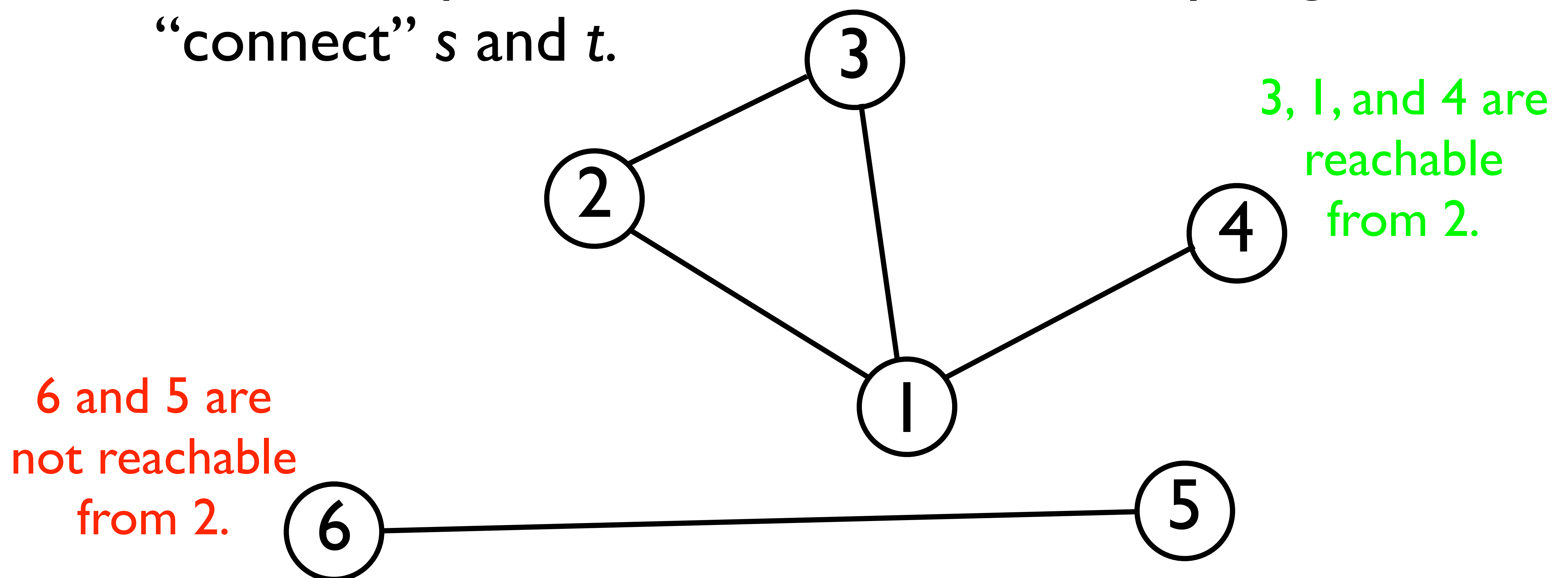
Node 1: { 2 }
Node 2: { 3 }
Node 3: { 1 }
Node 4: { 1 }
Node 5: { 4, 1 }

Graphs in computer science

- Graphs find many uses in computer science in almost every sub-discipline:
  - Computability/complexity theory.
  - Networking.
  - Machine learning.
  - Social networks.
  - Compilers
  - ...

Node discovery and shortest paths

- Two fundamental problems in graph theory are *node discovery* and finding a *shortest path* between two nodes.
- *Node discovery* is the process of determining the set of nodes **reachable** from a starting node s.
  - Informally, we say a node *t* is reachable from *s* if there exists a sequence of nodes connected by edges that "connect" *s* and *t*.



Node discovery and shortest paths

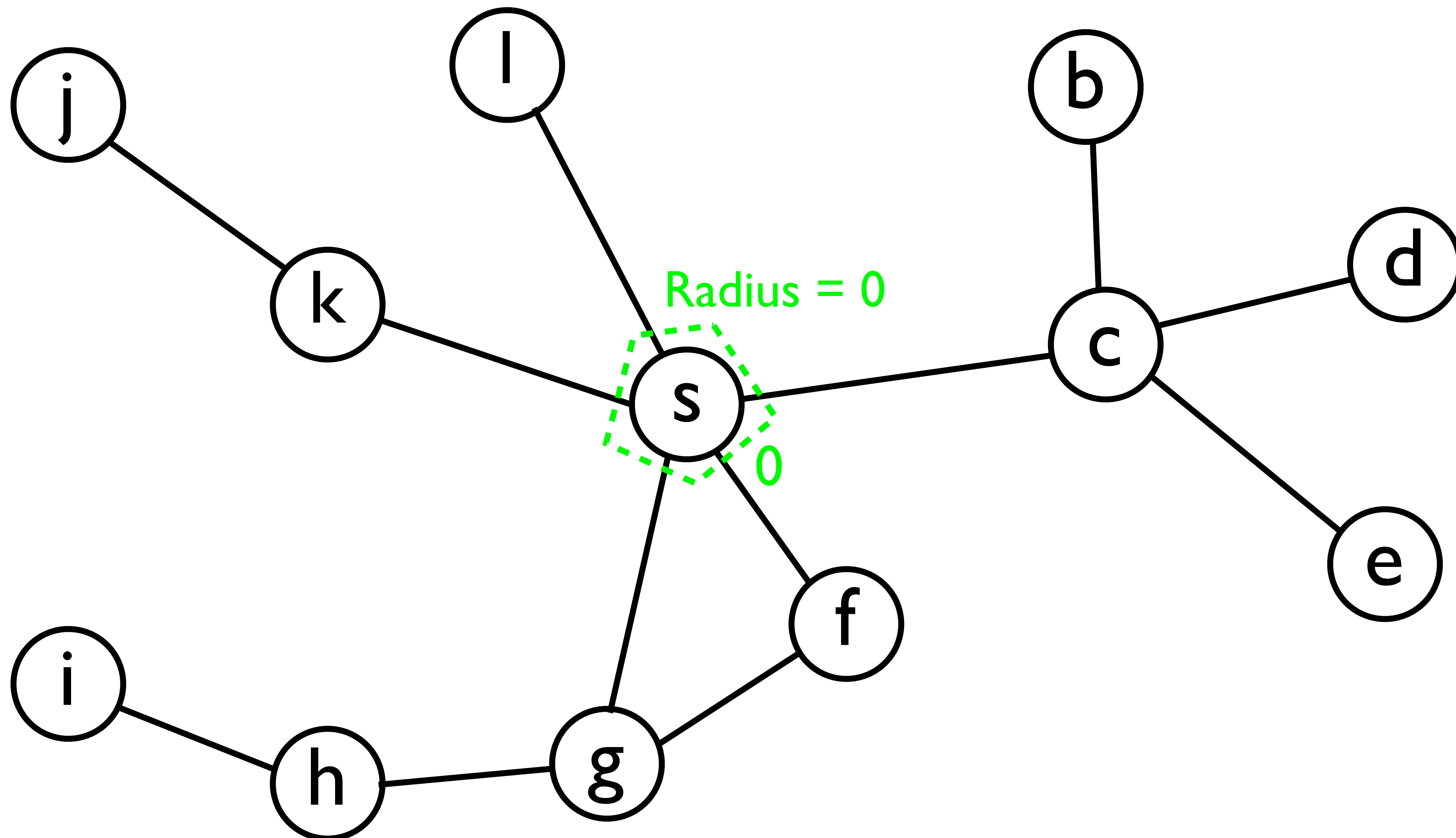
- Node discovery is important because it provides a way of *iterating* over all nodes in a graph that are reachable from some starting node.
- For instance, given a graph of cities (nodes) connected by bike trails (edges), we may wish to find the set of all cities that are *reachable* from La Jolla, CA by bicycle.
- The two principal algorithms for node discovery are Breadth-First-Search (BFS) and Depth-First-Search (DFS).
- Using BFS, we can also find a **shortest path** between two nodes *s* and *t*.

BFS and DFS

- BFS and DFS both solve the problem of discovering all nodes reachable from *s*.
- These algorithms differ in the *order* in which they discover/visit nodes:
  - BFS discovers/visits nodes by searching nodes within a fixed radius *r* that gradually increases:
    - First, search for all nodes 1 step away from *s*.
    - Next, search for all nodes 2 steps away from *s*.
    - Then, search for all nodes 3 steps away ...

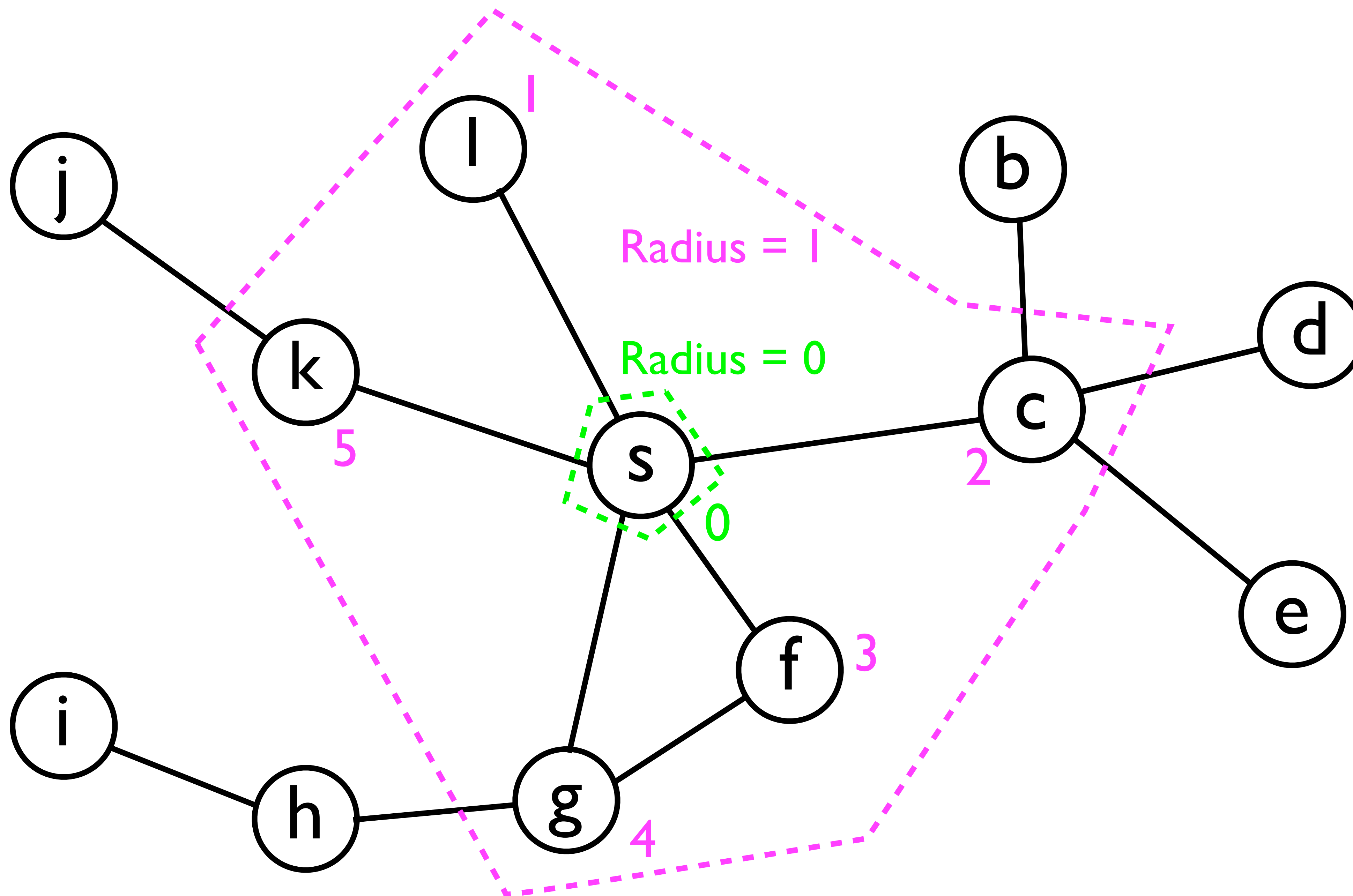
BFS

A BFS starts with at a node *s* and "radiates outwards".



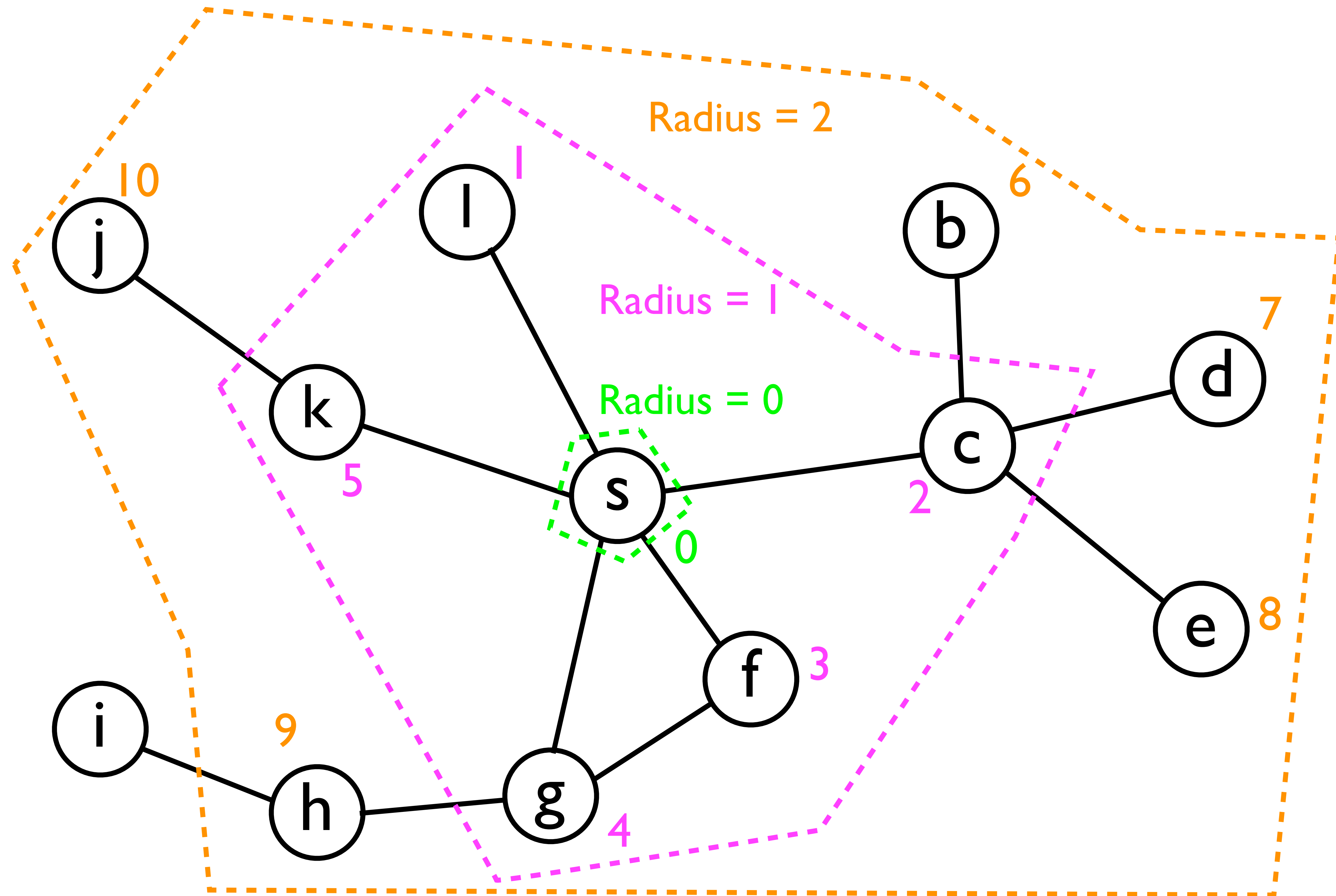
BFS

A BFS starts with at a node *s* and "radiates outwards".



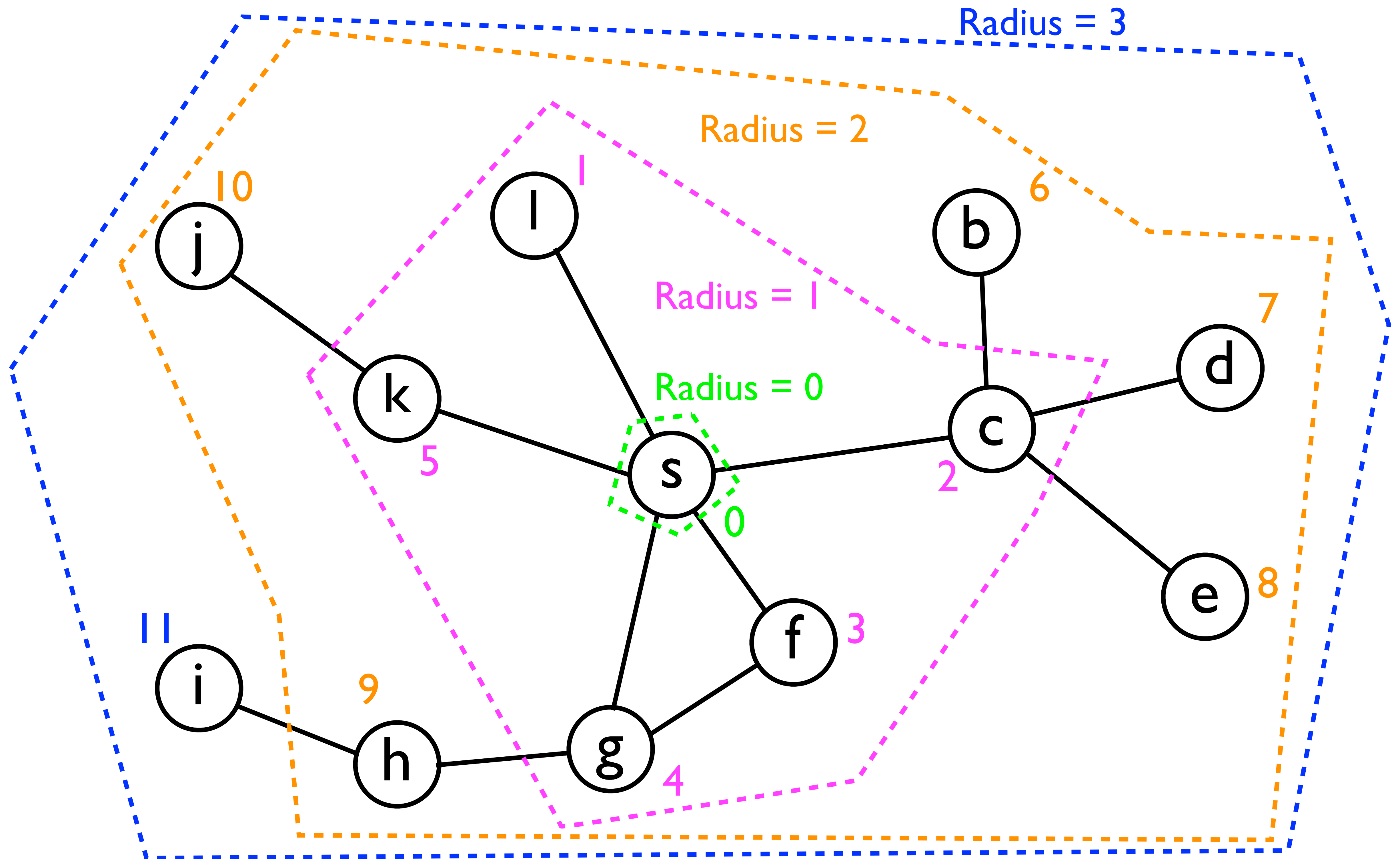
BFS

A BFS starts with at a node *s* and "radiates outwards".



BFS

A BFS starts with at a node *s* and "radiates outwards".

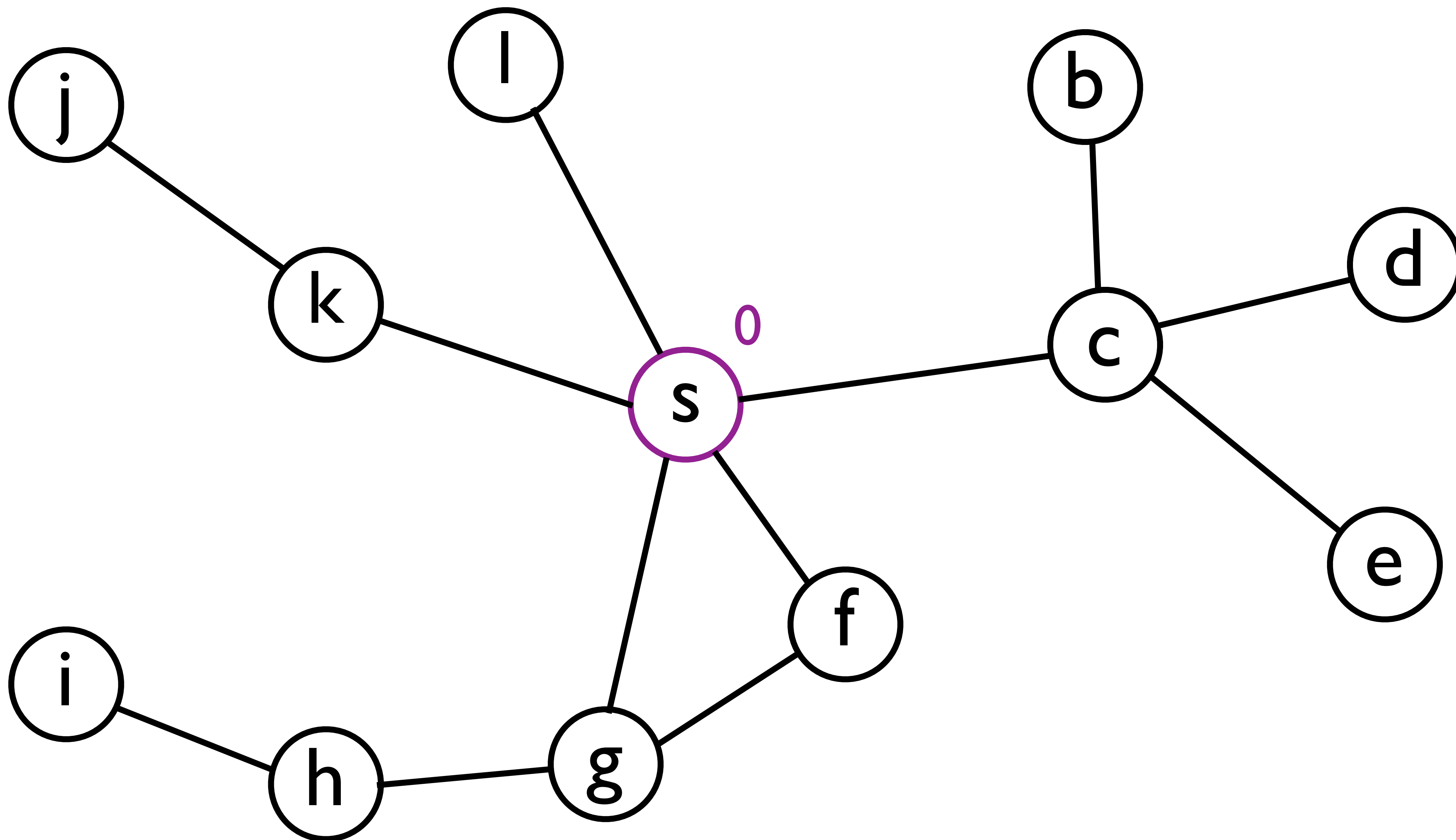


BFS and DFS

- In contrast, DFS discovers nodes by following one path away from *s* "as far as it can go".
  - When it reaches the "end" of a path, it *backtracks* and then follows another path "as far as it can go".

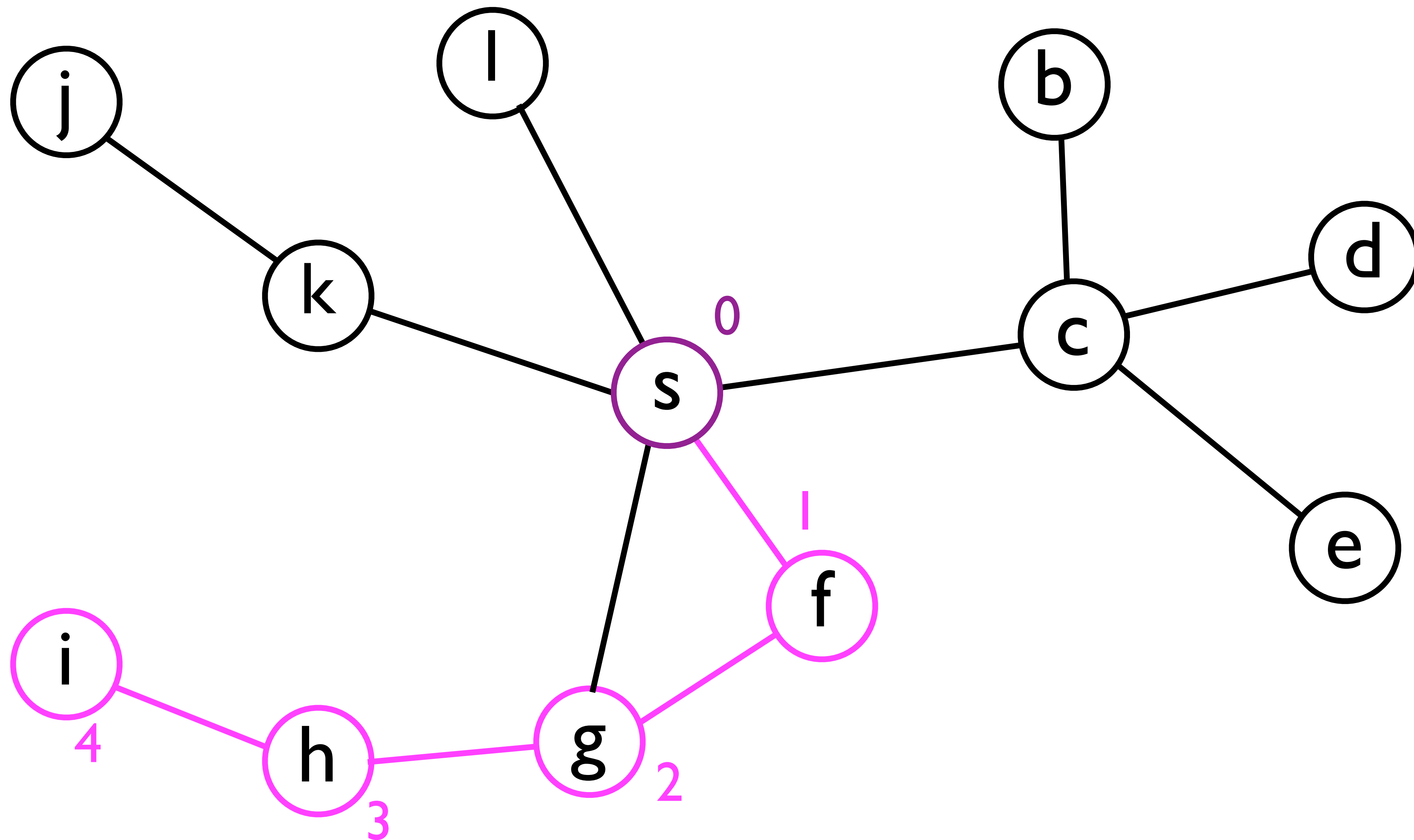
DFS

A DFS starts with at a node *s* and "goes as far as it can" down a single path.



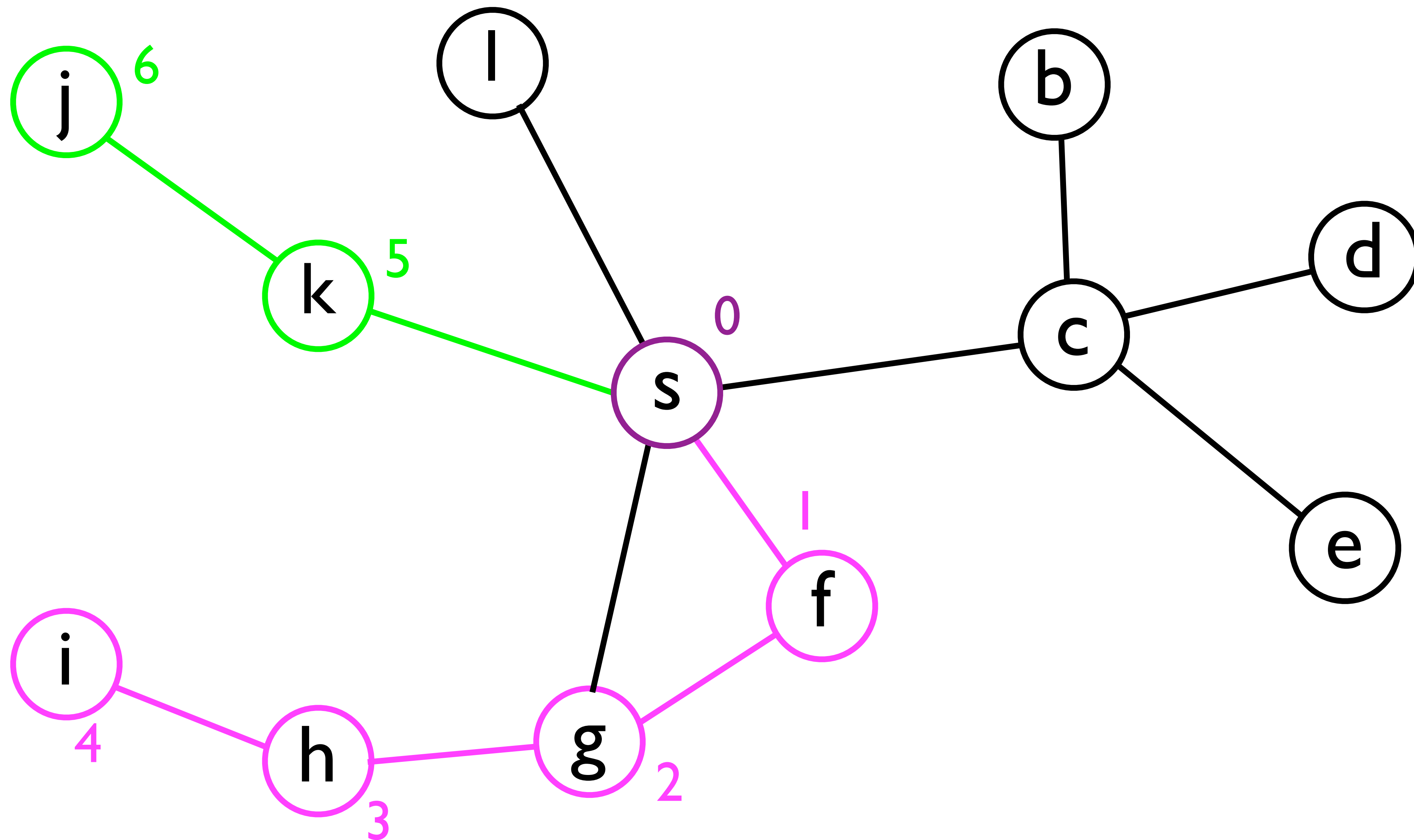
DFS

A DFS starts with at a node *s* and "goes as far as it can" down a single path.



DFS

At the end of a path, it then "backtracks" and pursues another path "as far as it can go".

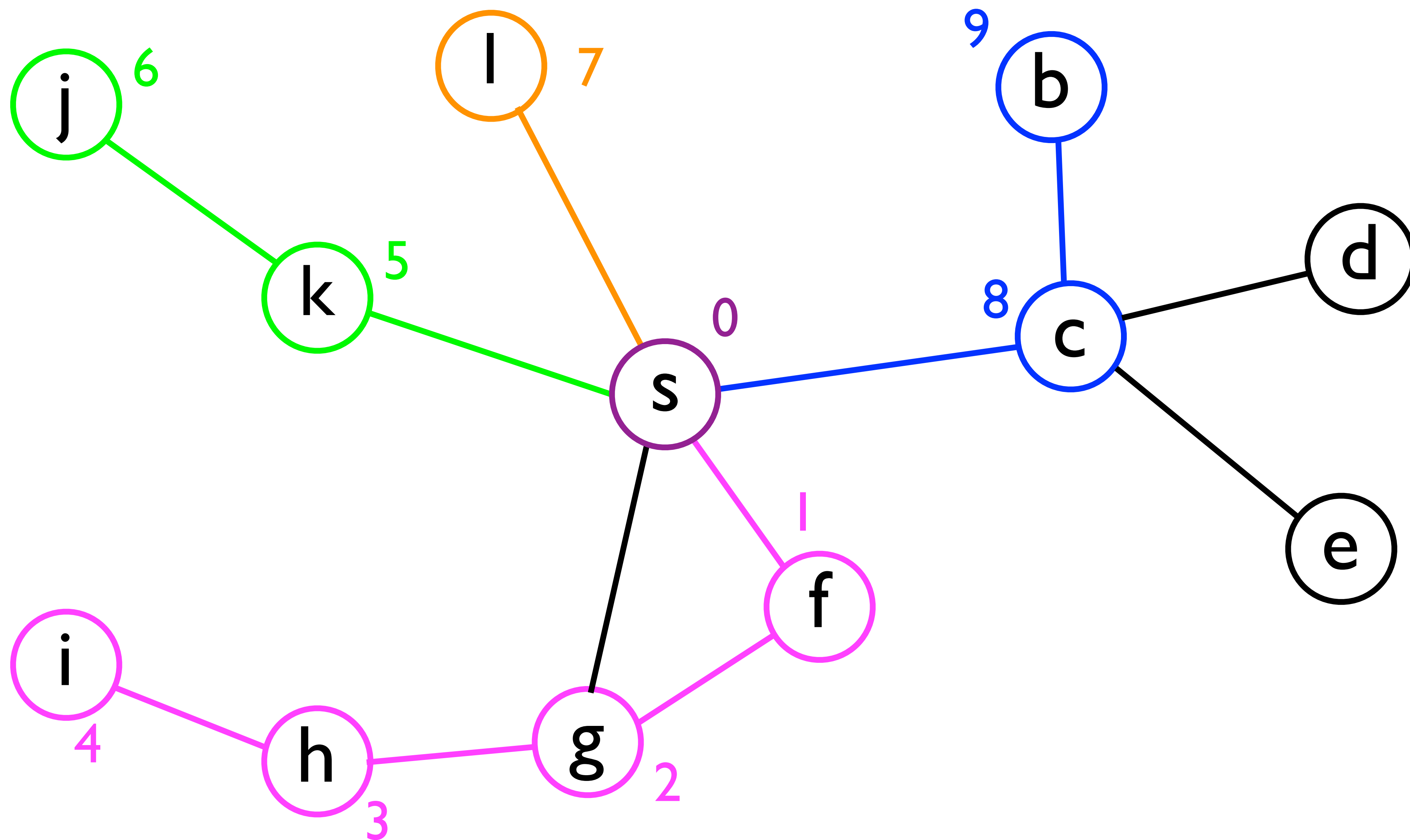


DFS

At the end of a path, it then "backtracks" and pursues another path "as far as it can go".

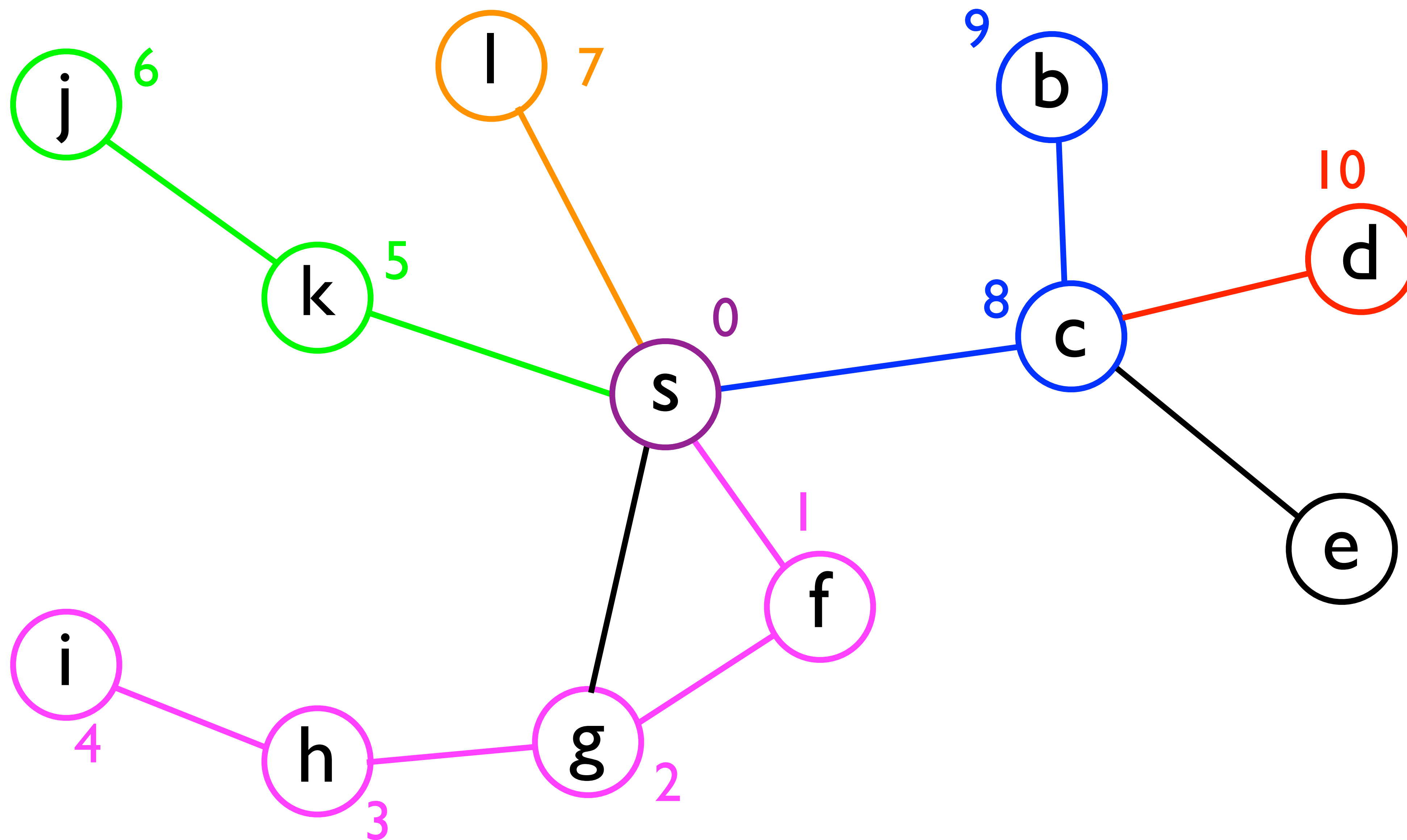
DFS

At the end of a path, it then "backtracks" and pursues another path "as far as it can go".



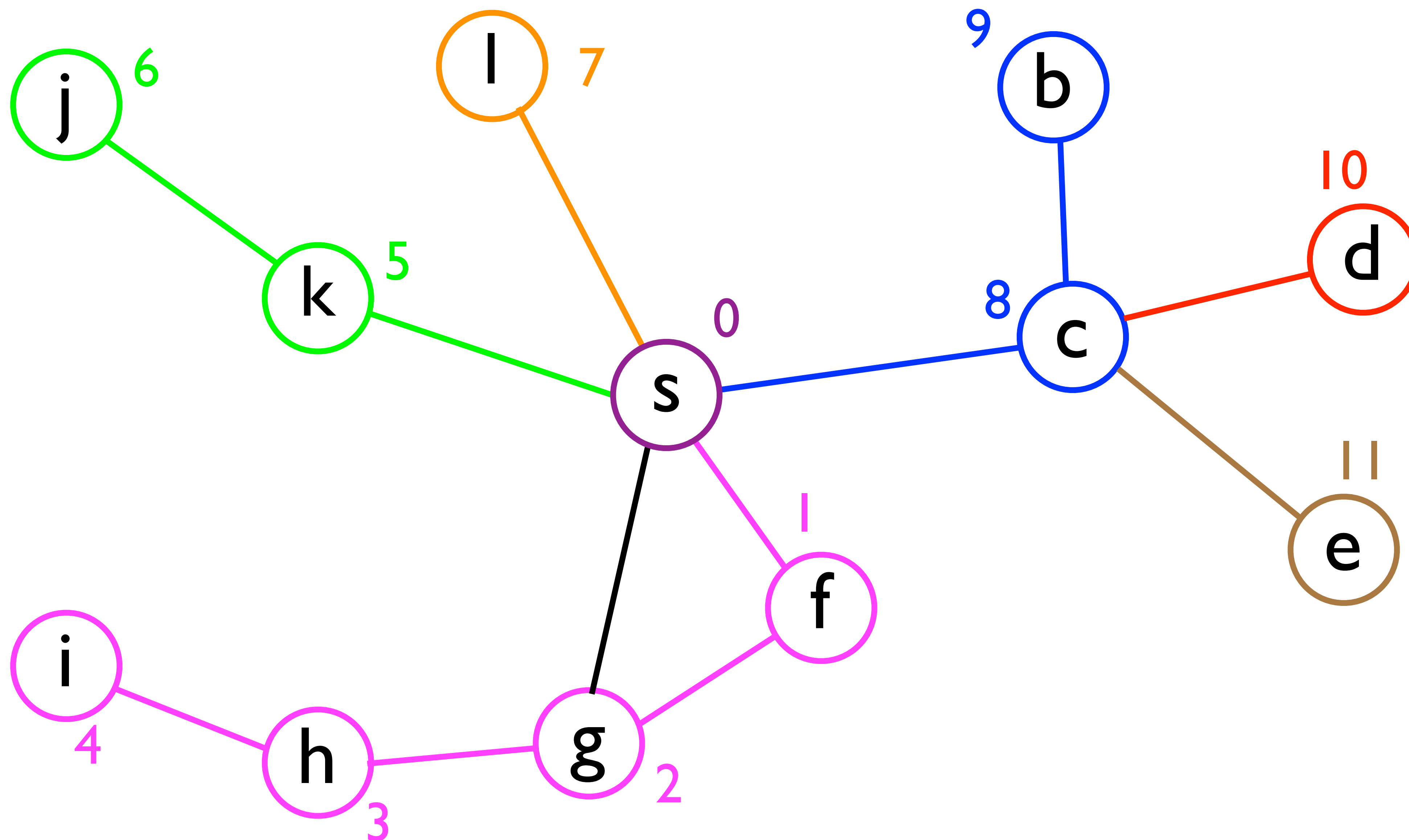
DFS

At the end of a path, it then "backtracks" and pursues another path "as far as it can go".



DFS

At the end of a path, it then “backtracks” and pursues another path “as far as it can go”.



Implementing a graph

- Before describing the BFS and DFS algorithms in detail, it will be useful to define some "infrastructure" for dealing with graphs.
- Let's suppose there exists a **`Node`** class to contain whatever is relevant for the user's application, e.g.:

```
static class Node {
  String _name;
  Image _facePic;
  int _age;
  Node[] _friends;  // adjacency list
}
```

- Notice how each **`Node`** contains a list of "friends" -- this is an *adjacency list* for that person.

Implementing a graph

- We will then define the whole **`Graph`** as follows:

```
class Graph {
  static class Node {
    String _name;
    Image _facePic;
    int _age;
    Node[] _friends;  // adjacency list
  }

  Node[] _people;
}
```

BFS implementation

- Given this graph infrastructure, we can start to define our **`bfs(s)`** method:
- Internally, **`bfs(s)`** will maintain three data structures:
    - A *list* of nodes it has already visited (from *s*).
    - A *queue* of nodes it has yet to visit.

BFS implementation

- In pseudocode, **`bfs(s)`** looks as follows:

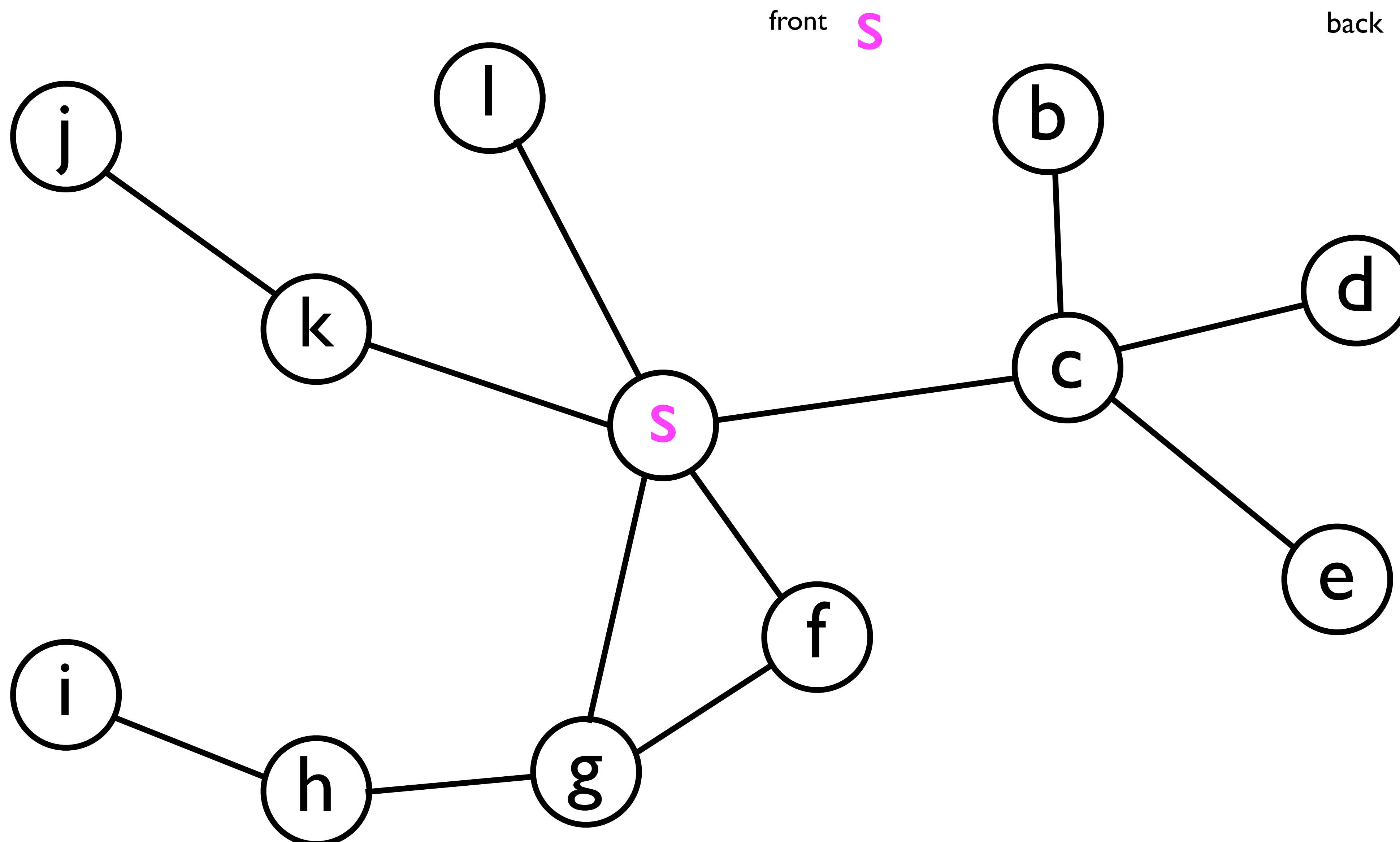
```
List<Node> bfs (Node s) {
  List<Node> visitedNodes;
  Queue<Node> nodesToVisit;

  nodesToVisit.enqueue(s);
  while (nodesToVisit.size() > 0) {
    n = nodesToVisit.dequeue();
    visitedNodes.add(n);

    for each friend n′ of n:
      if n′ not in nodesToVisit and n′ not in visitedNodes
        nodesToVisit.enqueue(n′);
  }

  return visitedNodes;
}
```

BFS in practice

- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

`nodesToVisit`:



BFS in practice

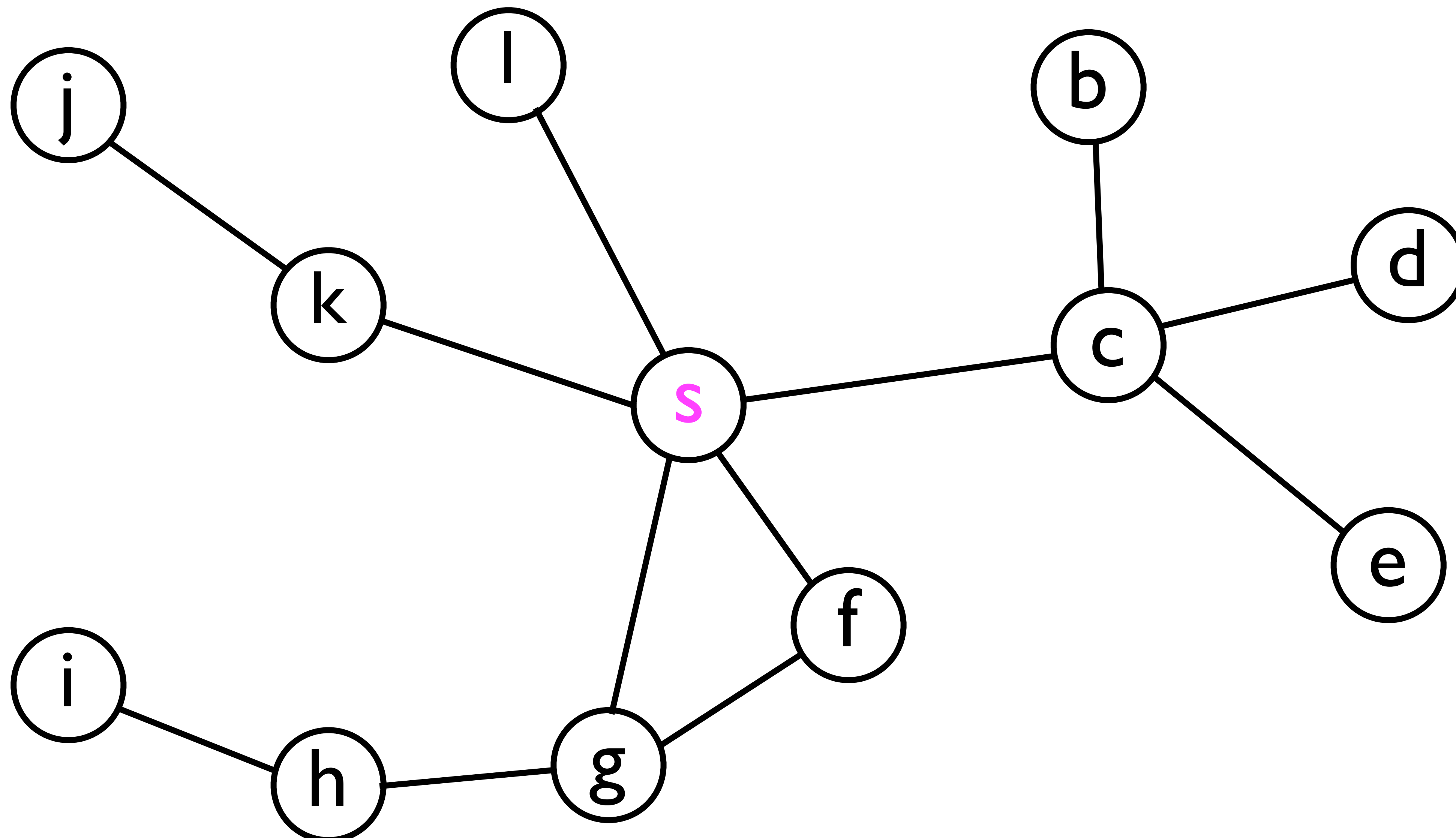
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s

`nodesToVisit`:

front back



BFS in practice

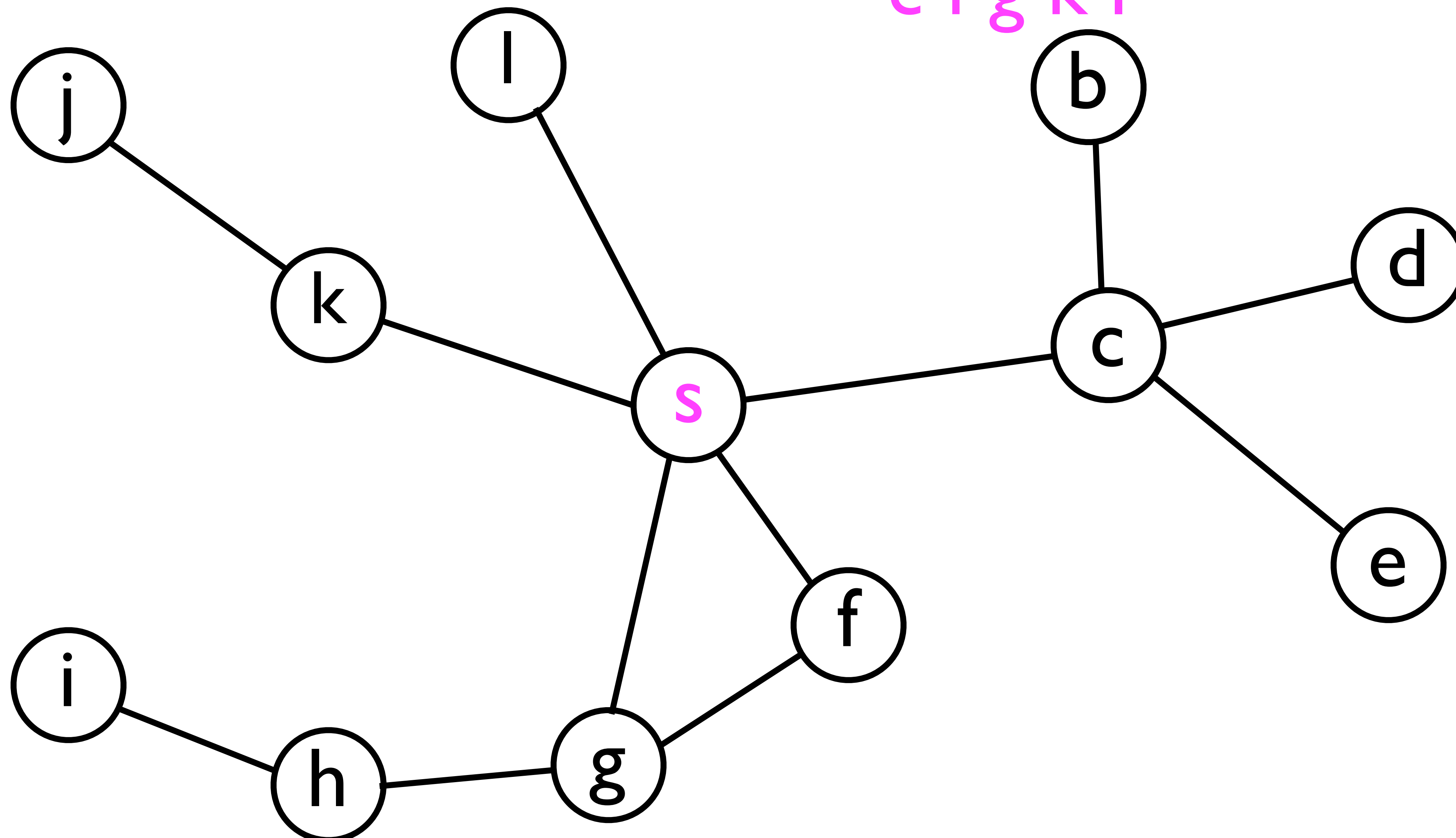
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s

`nodesToVisit`:

front c f g k l back



BFS in practice

- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes`:

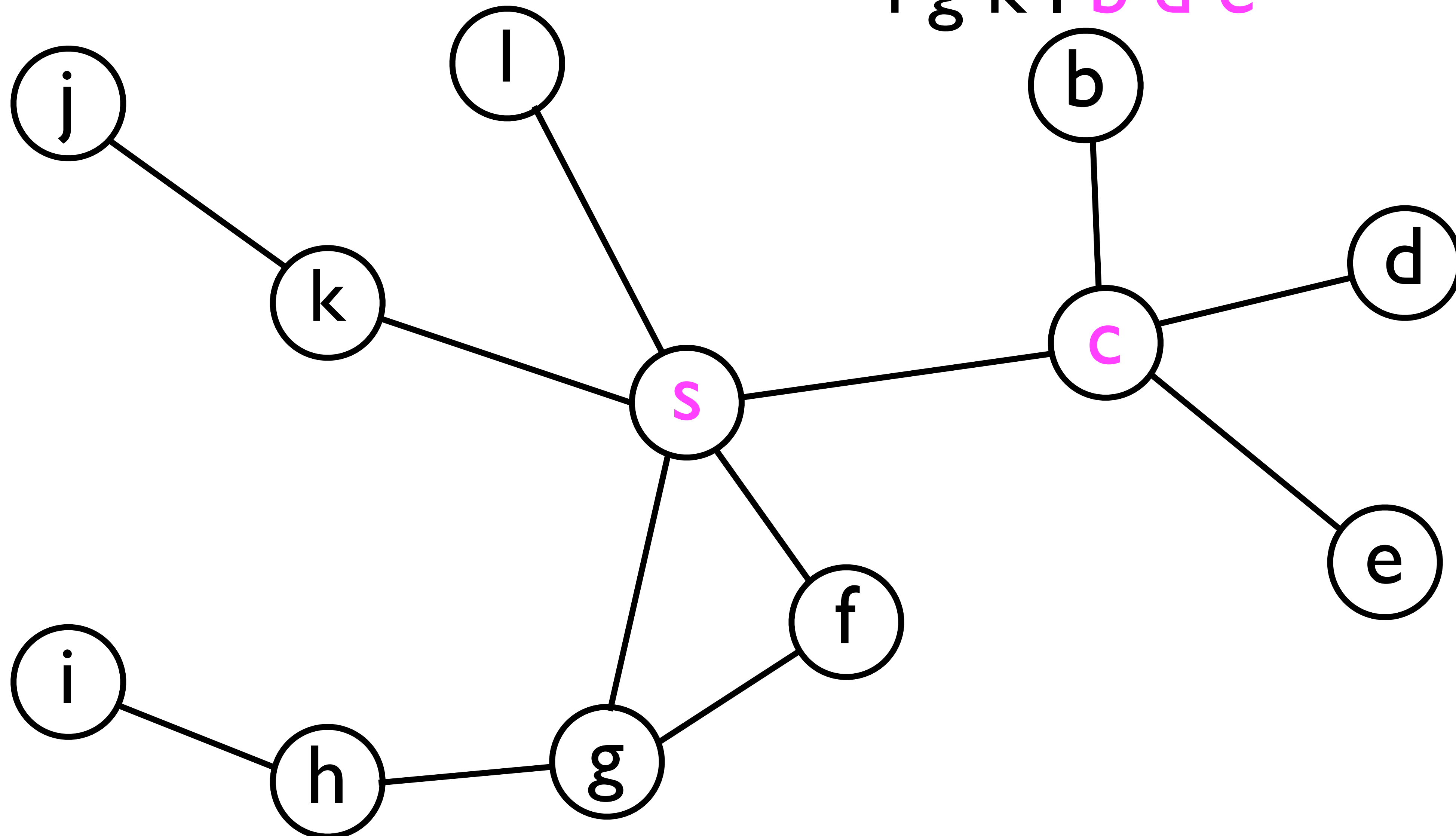
s c

`nodesToVisit`:

front f g k l back

BFS in practice

- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...



BFS in practice

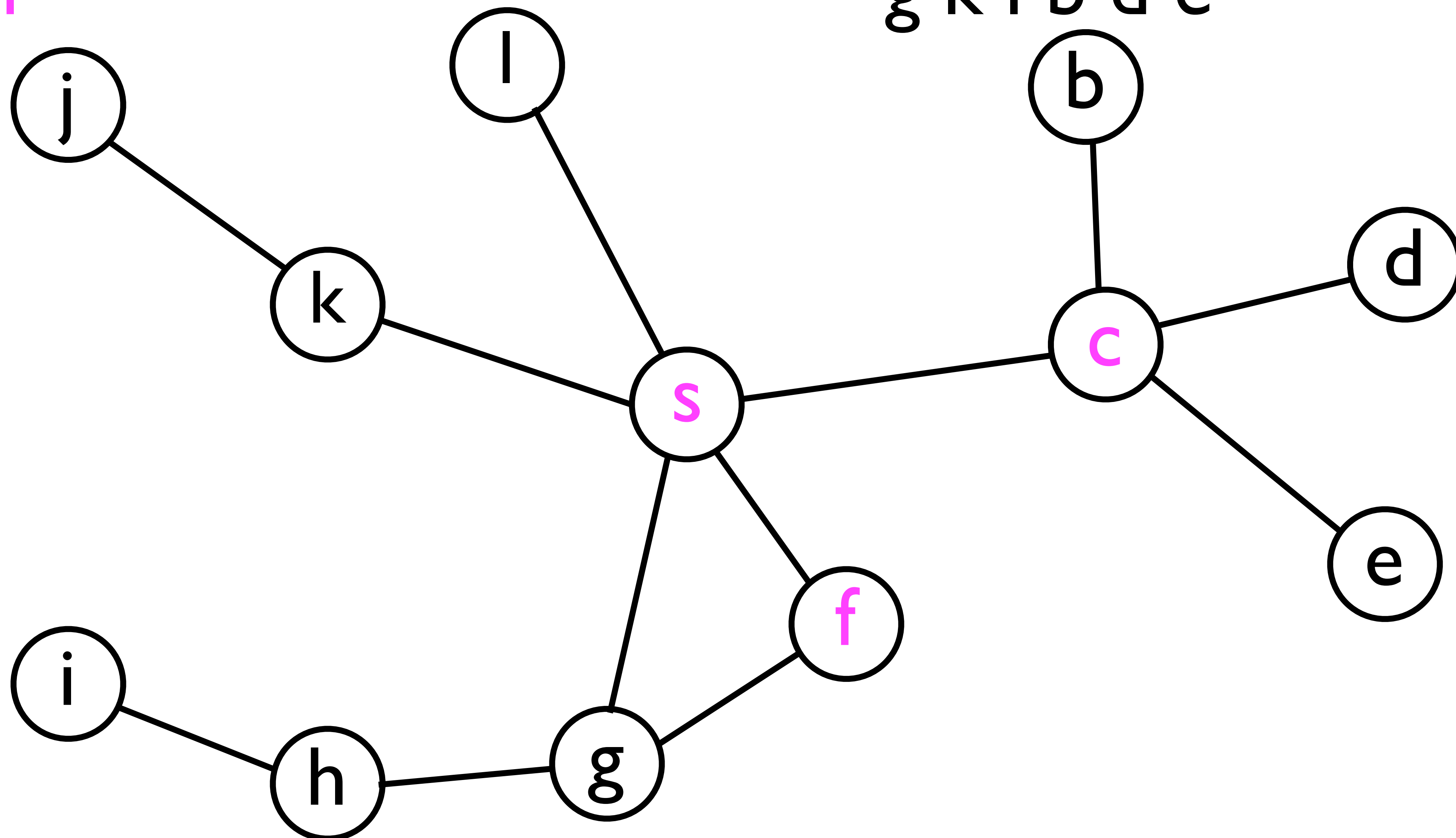
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f

`nodesToVisit`:

front g k l b d e back



BFS in practice

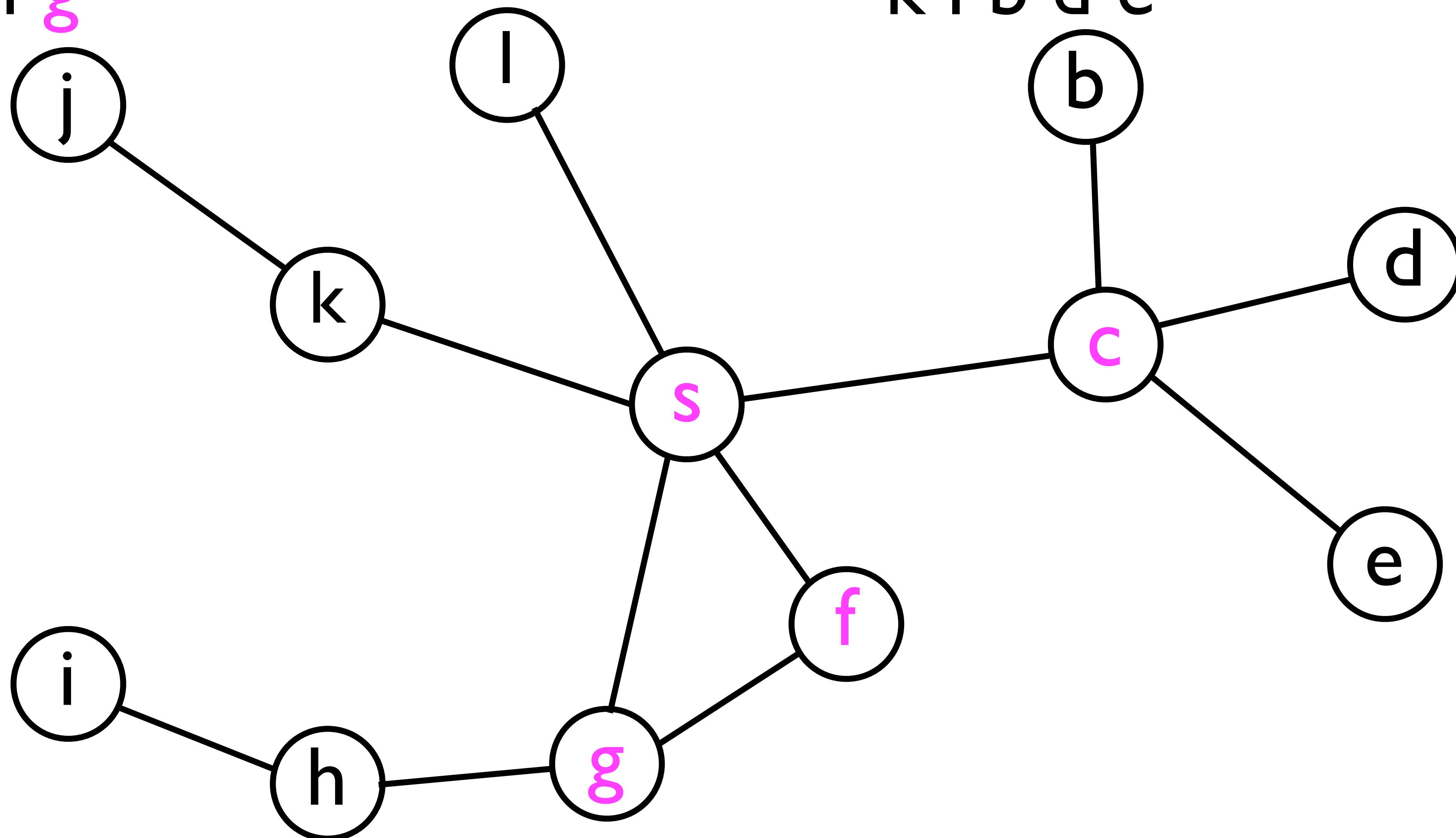
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g

`nodesToVisit`:

front k l b d e back



BFS in practice

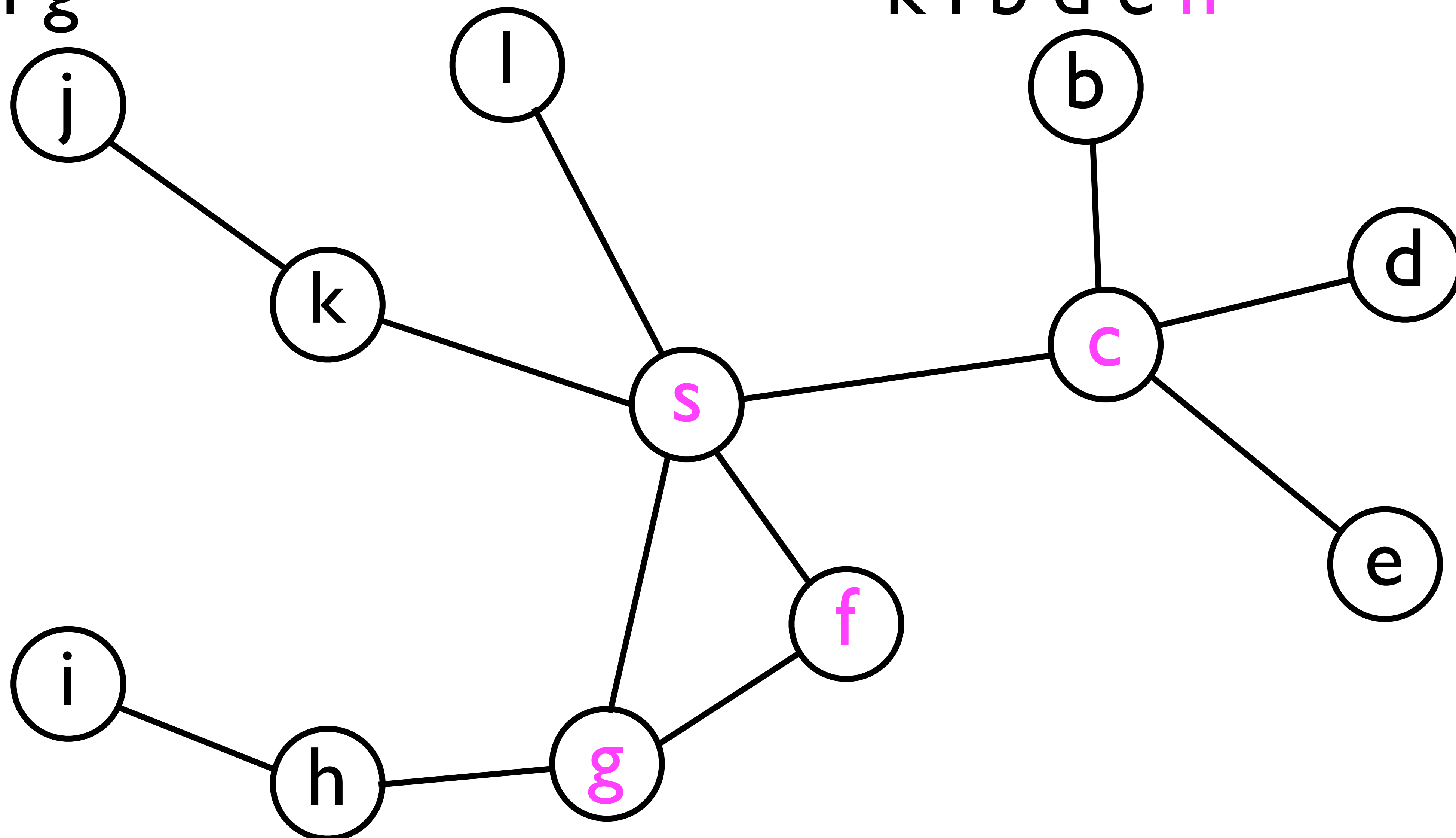
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g

`nodesToVisit`:

front k l b d e h back



BFS in practice

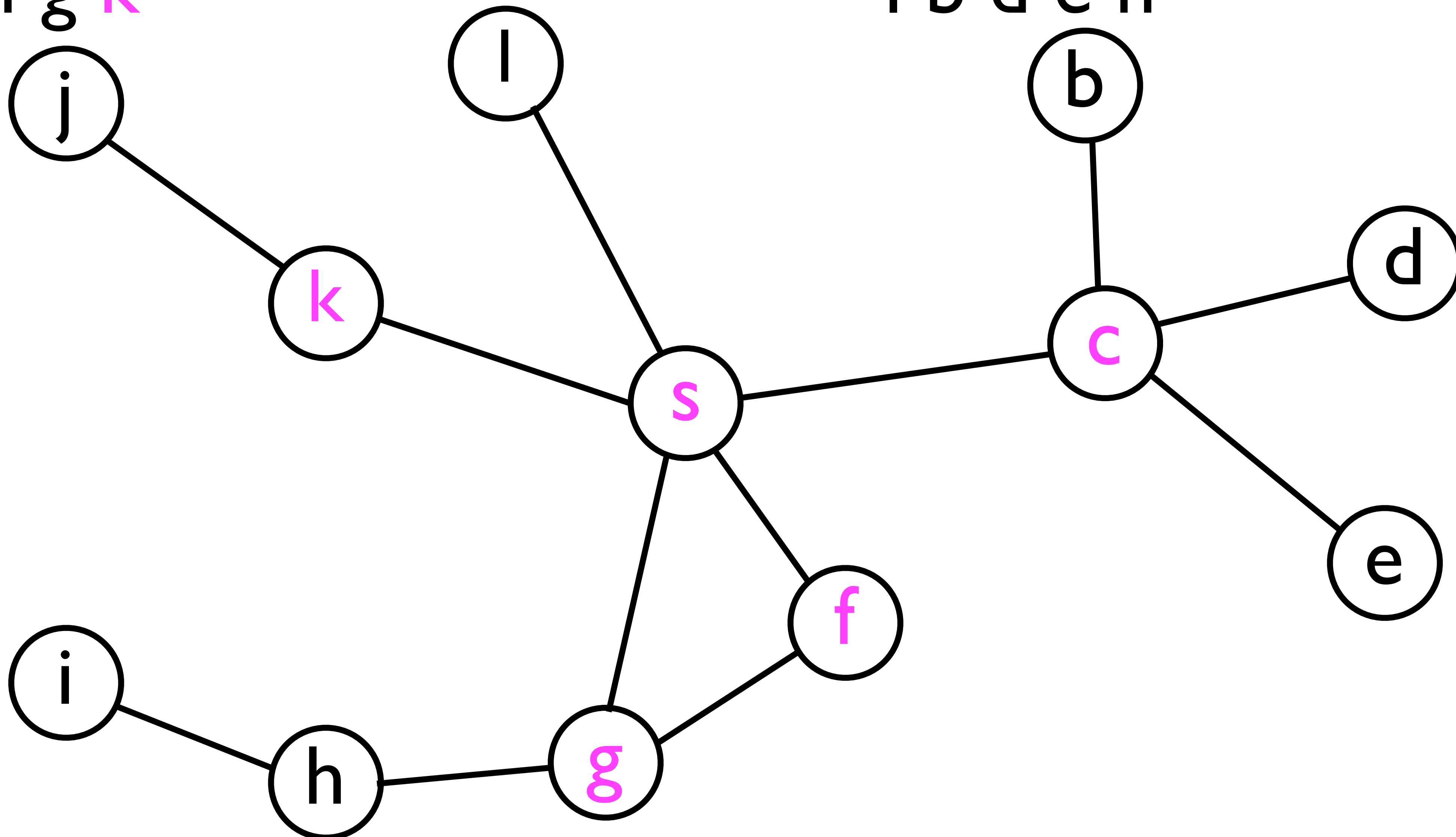
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k

`nodesToVisit`:

front l b d e h back



BFS in practice

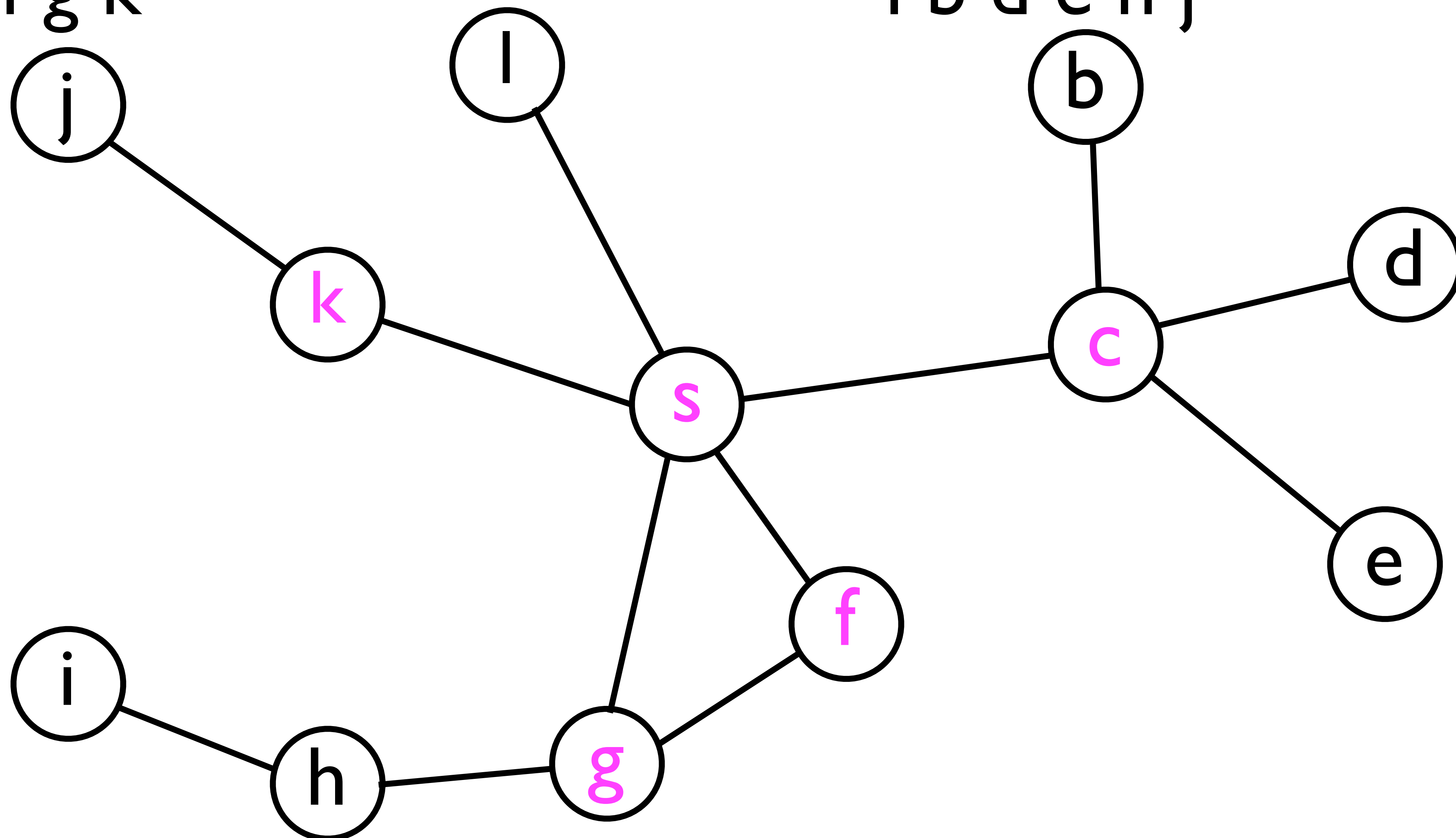
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k

`nodesToVisit`:

front l b d e h j back



BFS in practice

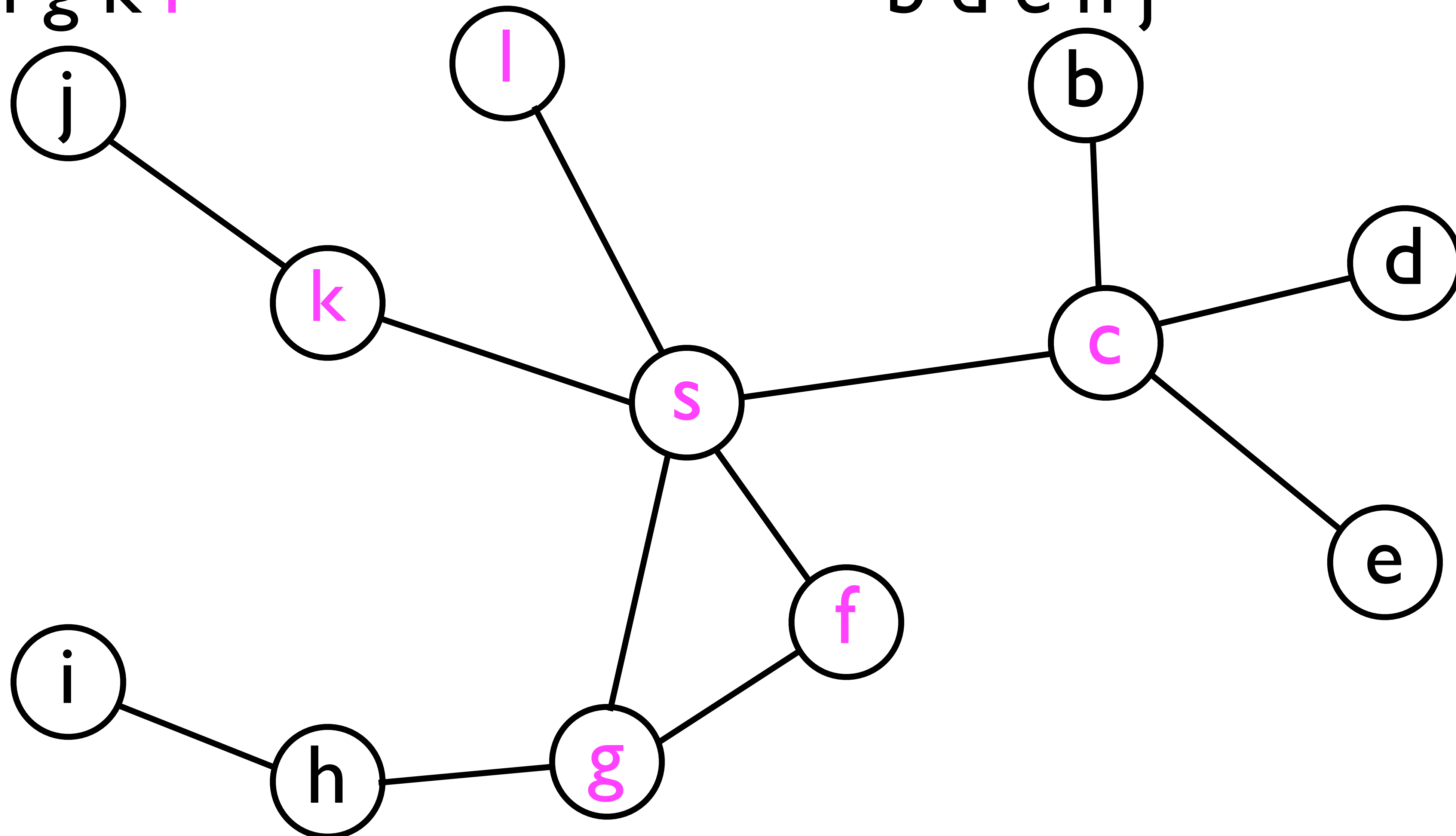
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k l

`nodesToVisit`:

front b d e h j back



BFS in practice

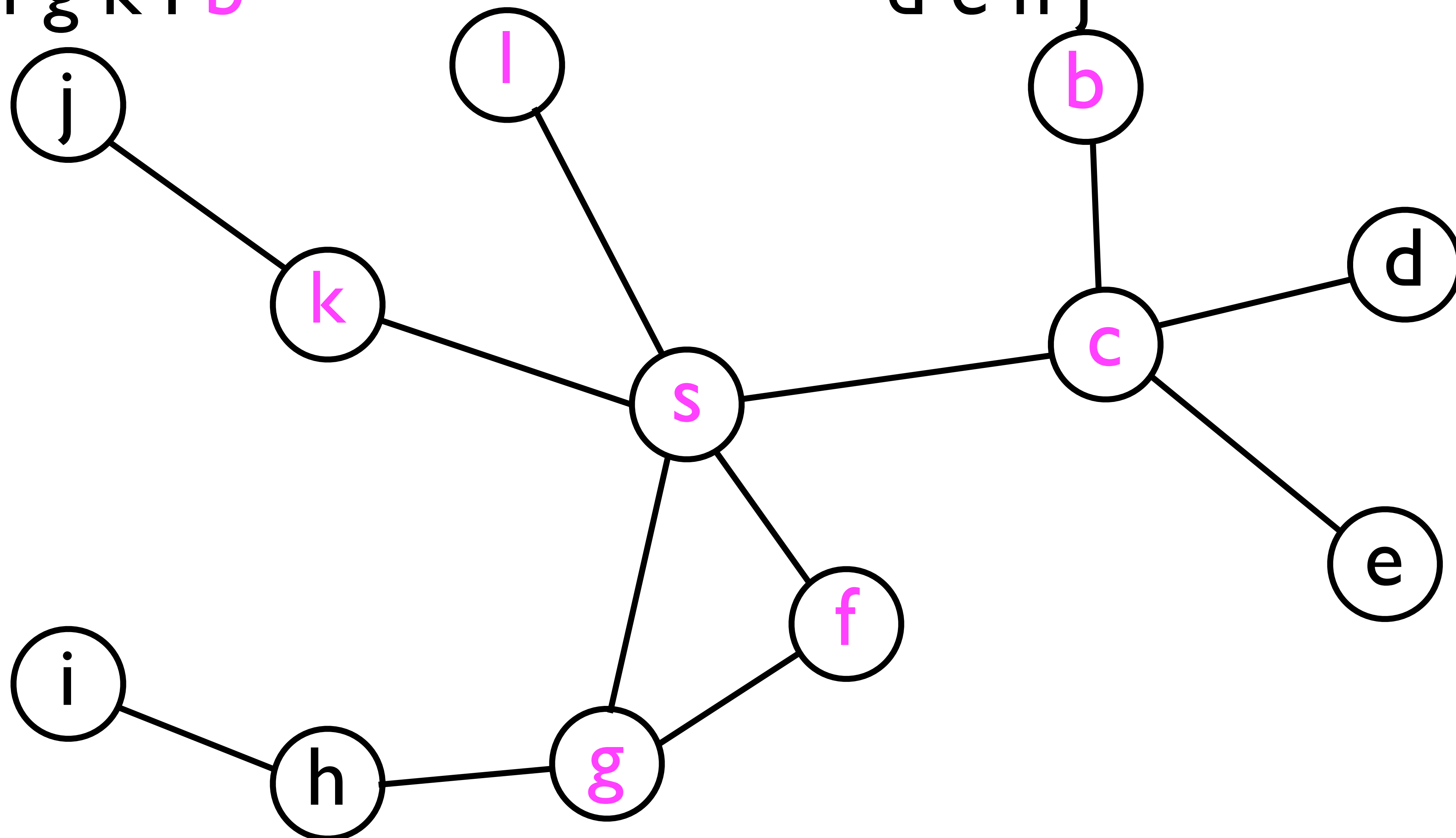
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k l b

`nodesToVisit`:

front d e h j back



BFS in practice

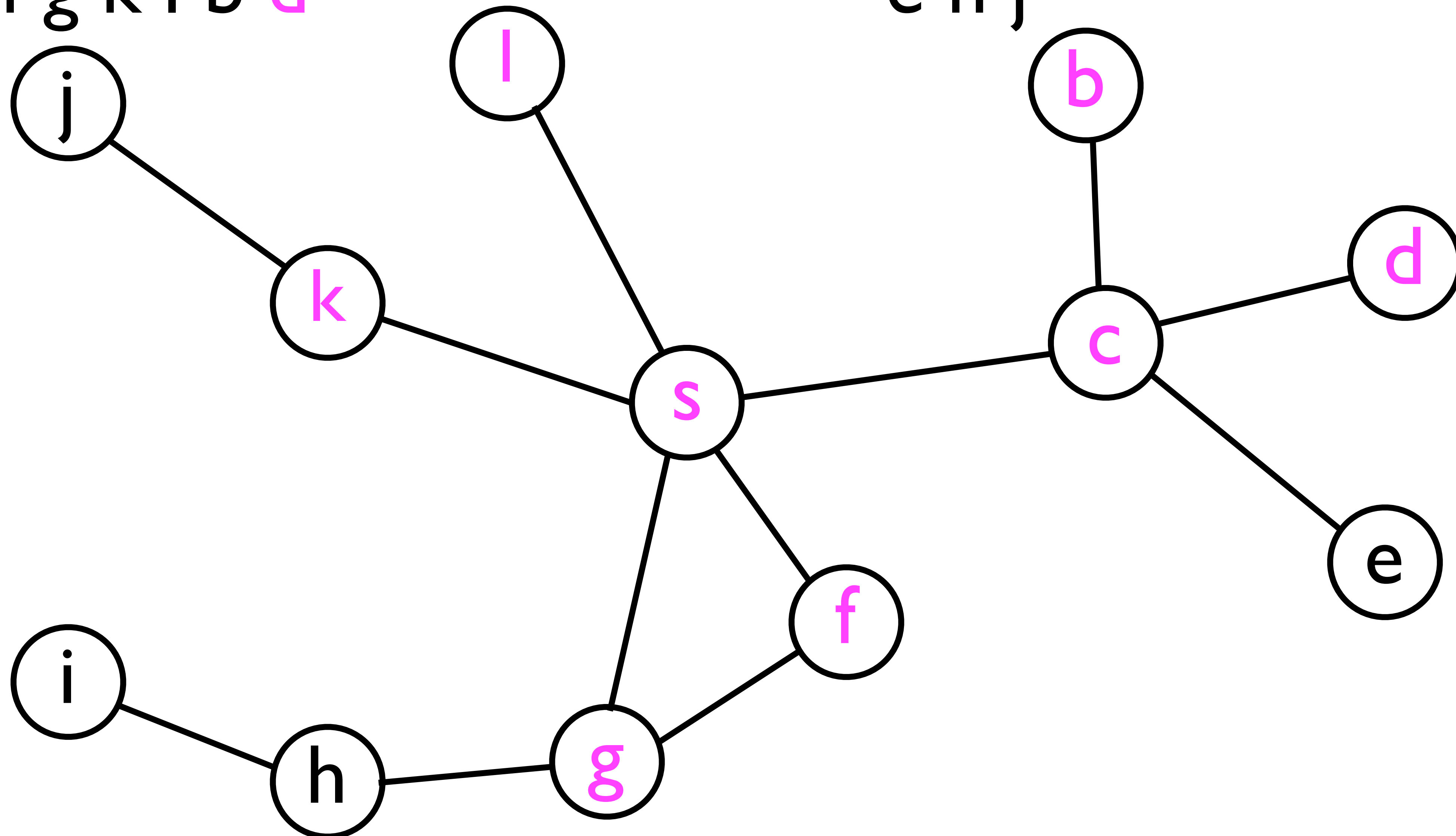
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k l b d

`nodesToVisit`:

front e h j back



BFS in practice

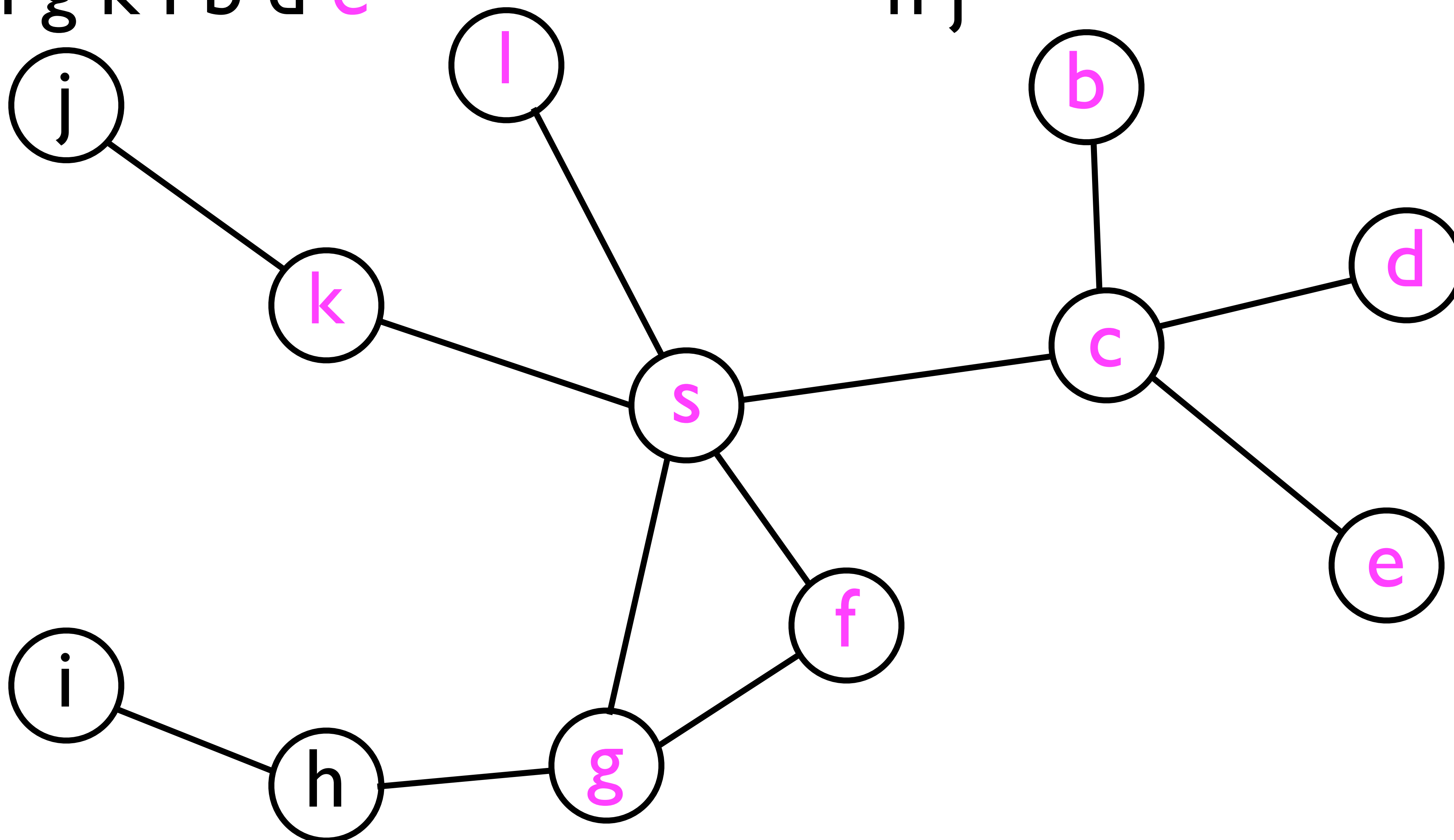
- Let’s look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k l b d e

`nodesToVisit`:

front h j back



BFS in practice

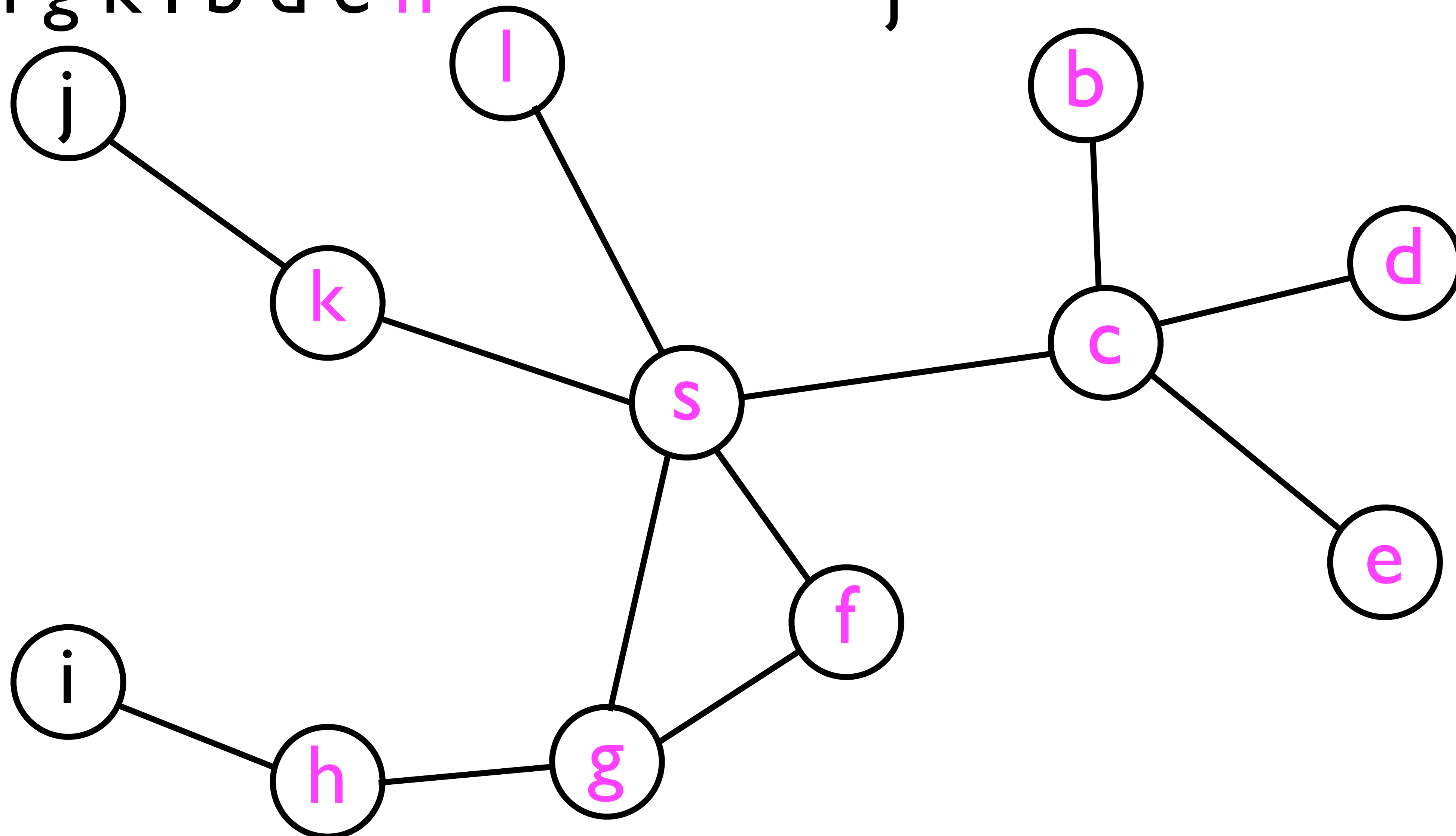
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k l b d e h

`nodesToVisit`:

front j back



BFS in practice

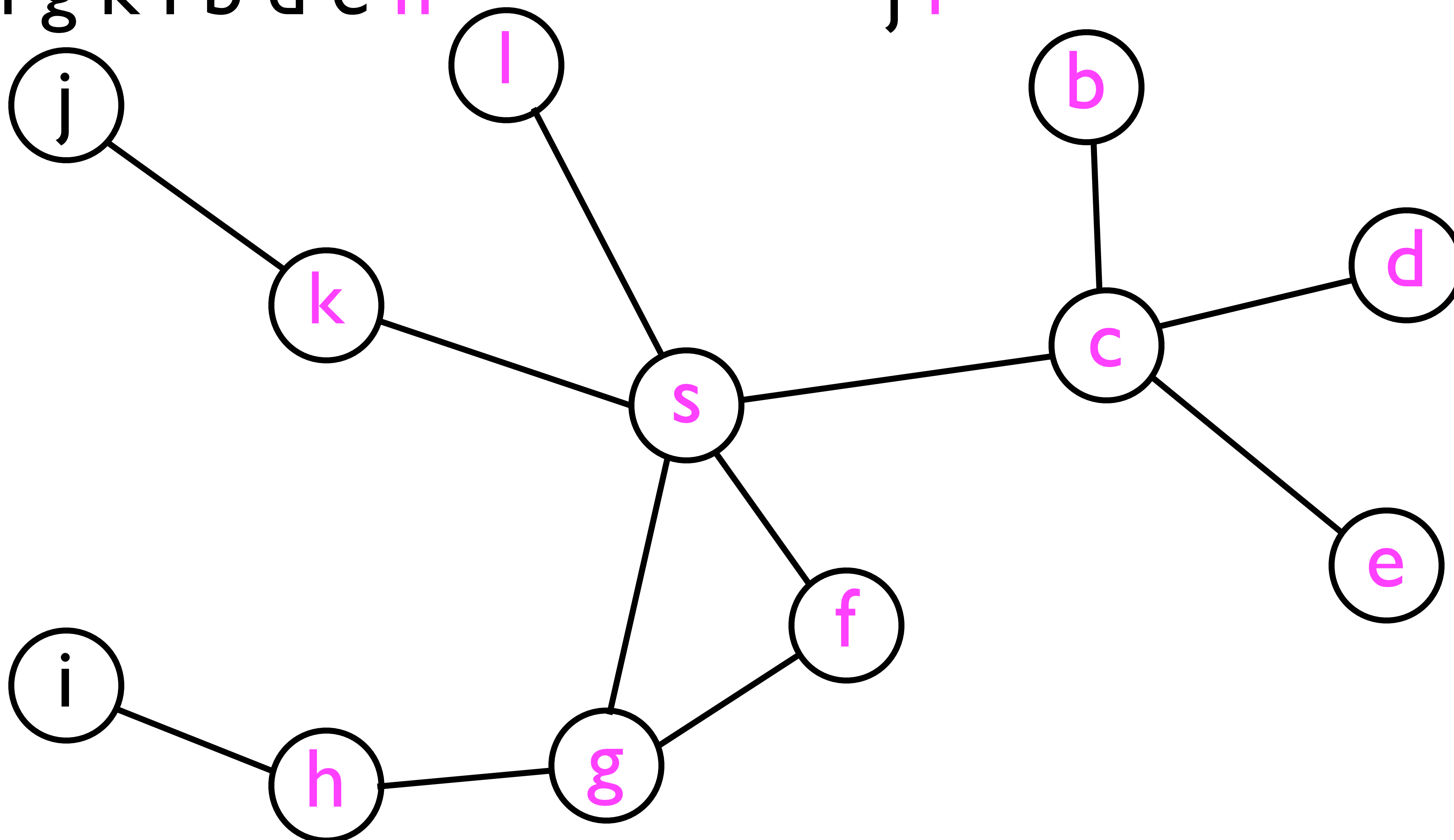
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k l b d e h

`nodesToVisit`:

front j i back



BFS in practice

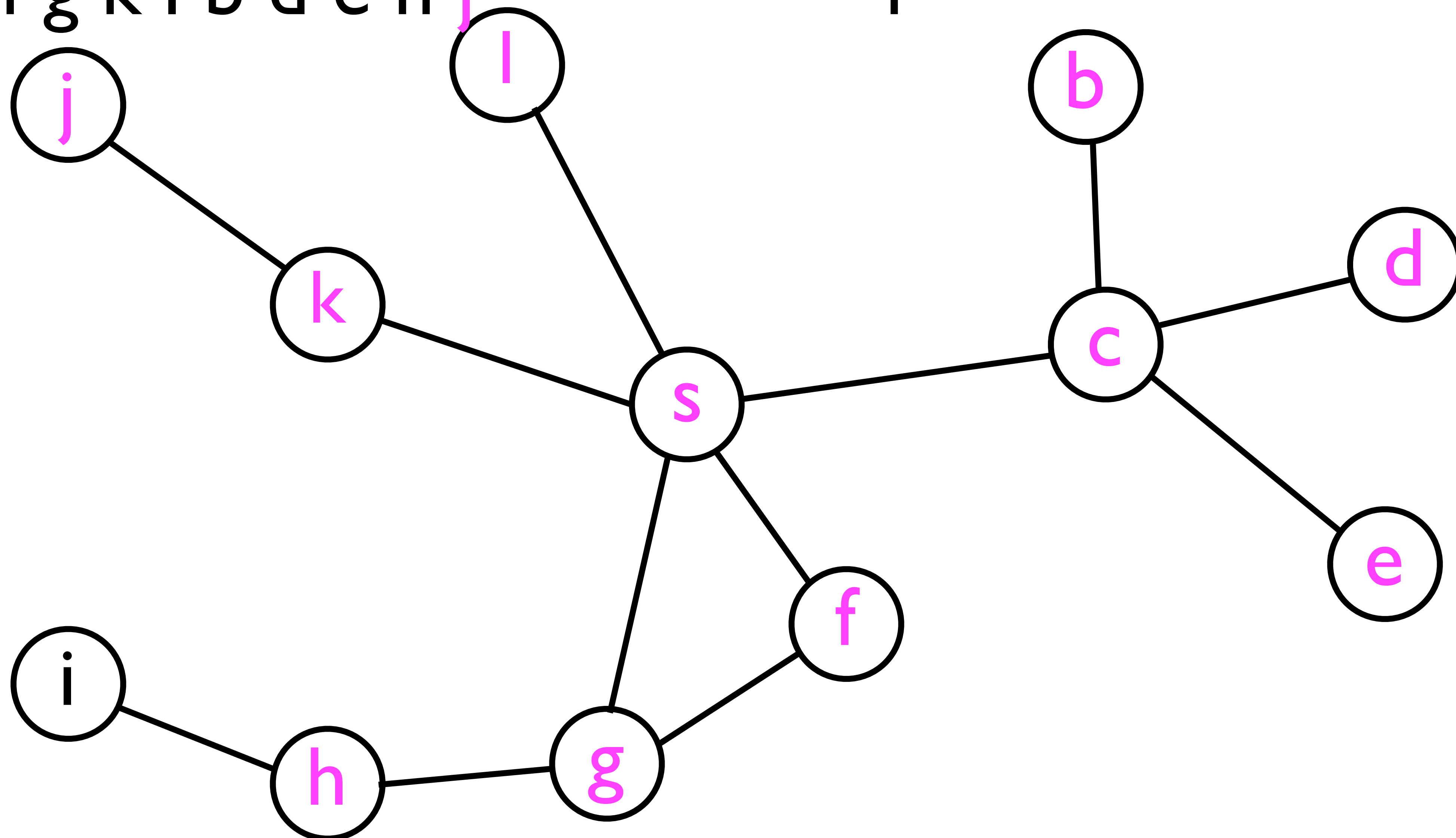
- Let’s look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k l b d e h j

`nodesToVisit`:

front i back



BFS in practice

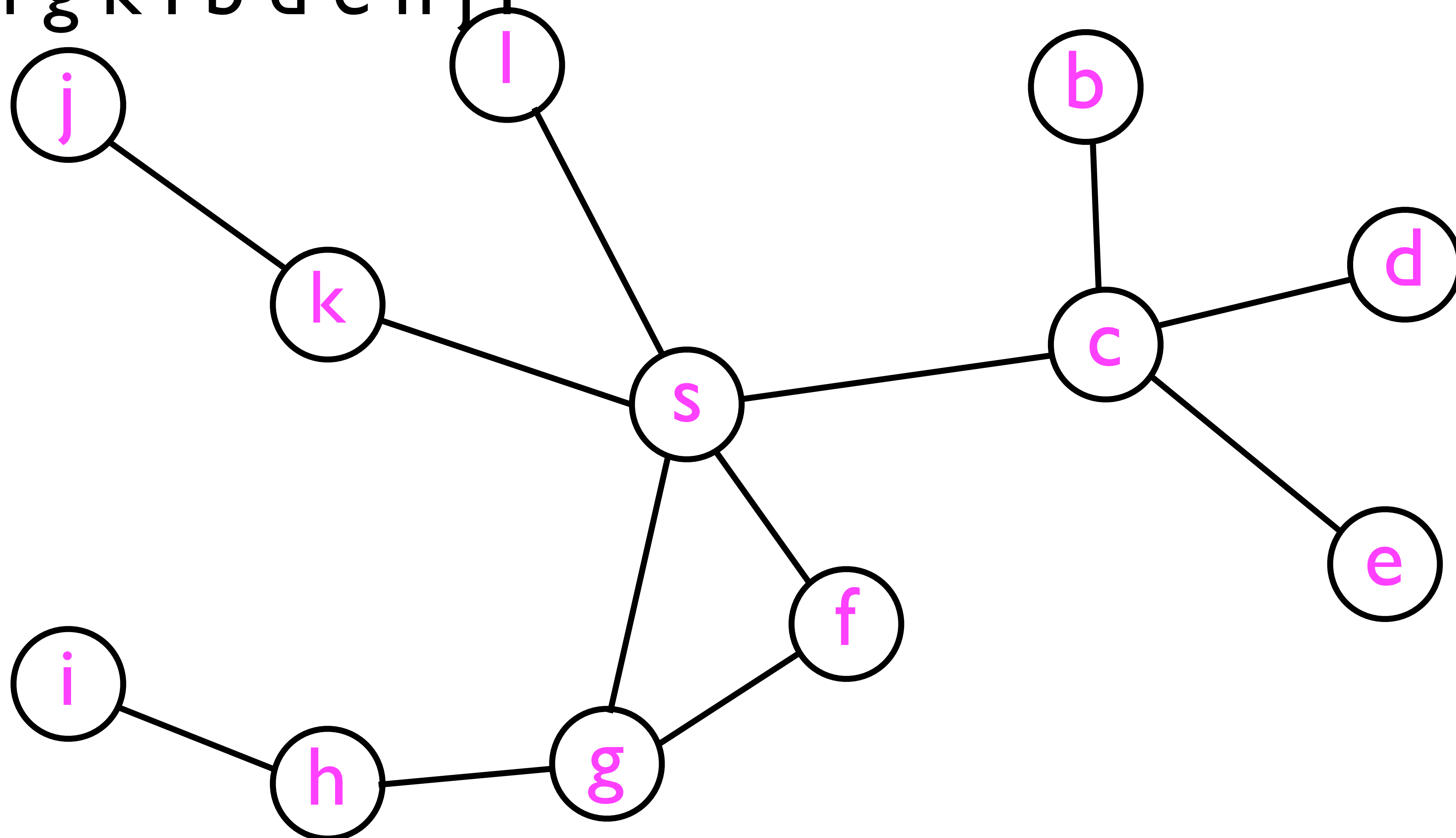
- Let’s look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s c f g k l b d e h j i

`nodesToVisit`:

front back



DFS implementation

- To implement *depth*-first-search (DFS), it turns out we can use the exact same pseudocode as for BFS *except* that we replace the *queue* with a *stack*.
- That this method works on the previous example will be left as an exercise to the reader.

DFS implementation

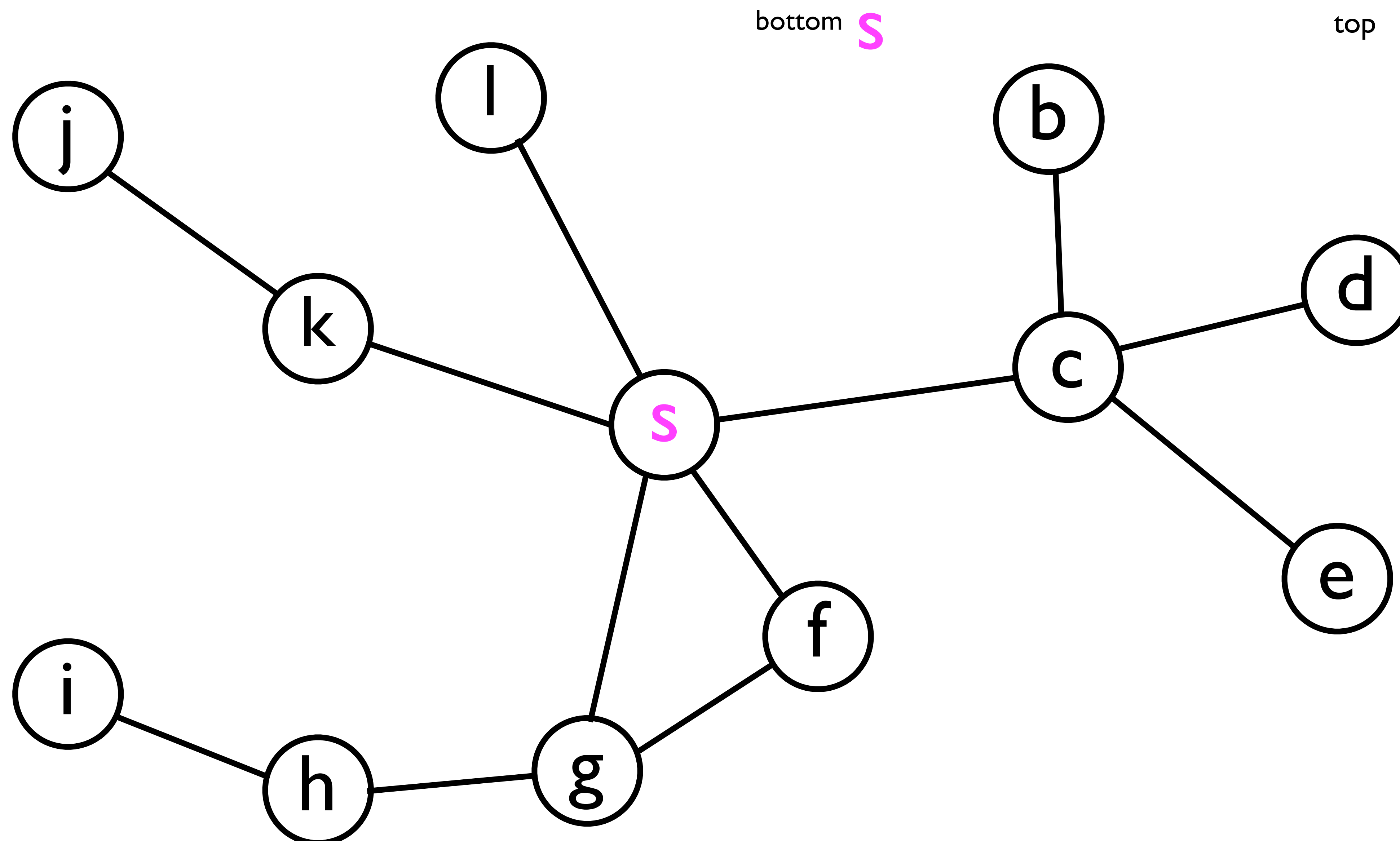
- In pseudocode, `dfs(s)` looks as follows:

```
List<Node> dfs (Node s) {
  List<Node> visitedNodes;
  Stack<Node> nodesToVisit;

  nodesToVisit.push(s);
  while (nodesToVisit.size() > 0) {
    n = nodesToVisit.pop();
    visitedNodes.add(n);

    for each friend n′ of n:
      if n′ not in nodesToVisit and n′ not in visitedNodes
        nodesToVisit.push(n′);
  }

  return visitedNodes;
}
```

DFS in practice

- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`: **`nodesToVisit`**:



DFS in practice

- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s

`nodesToVisit`:

bottom top

l
j
b
k
d
s
c
e
f
i
g
h

DFS in practice

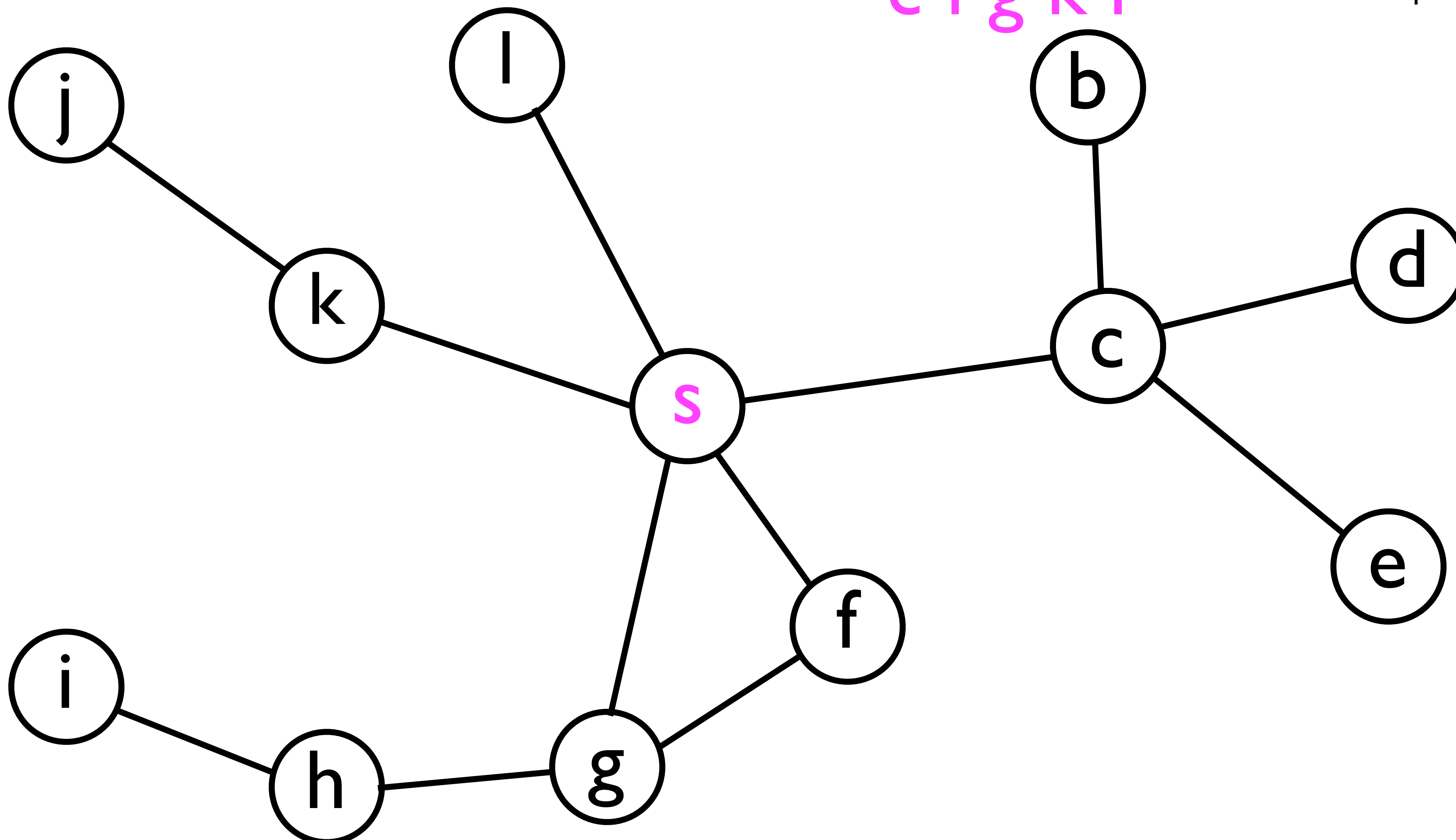
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s

`nodesToVisit`:

bottom c f g k l top



DFS in practice

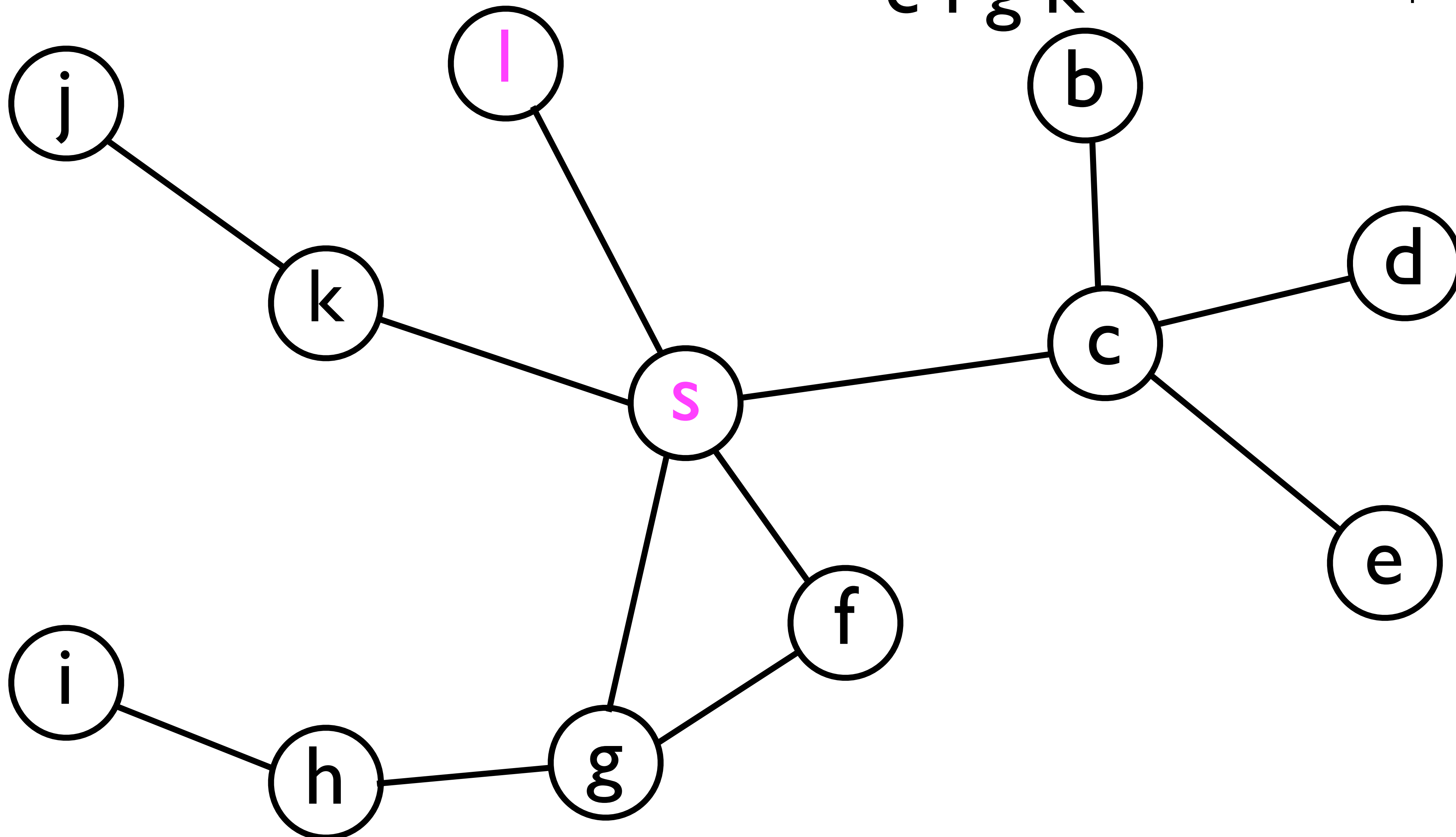
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s l

`nodesToVisit`:

bottom c f g k top



DFS in practice

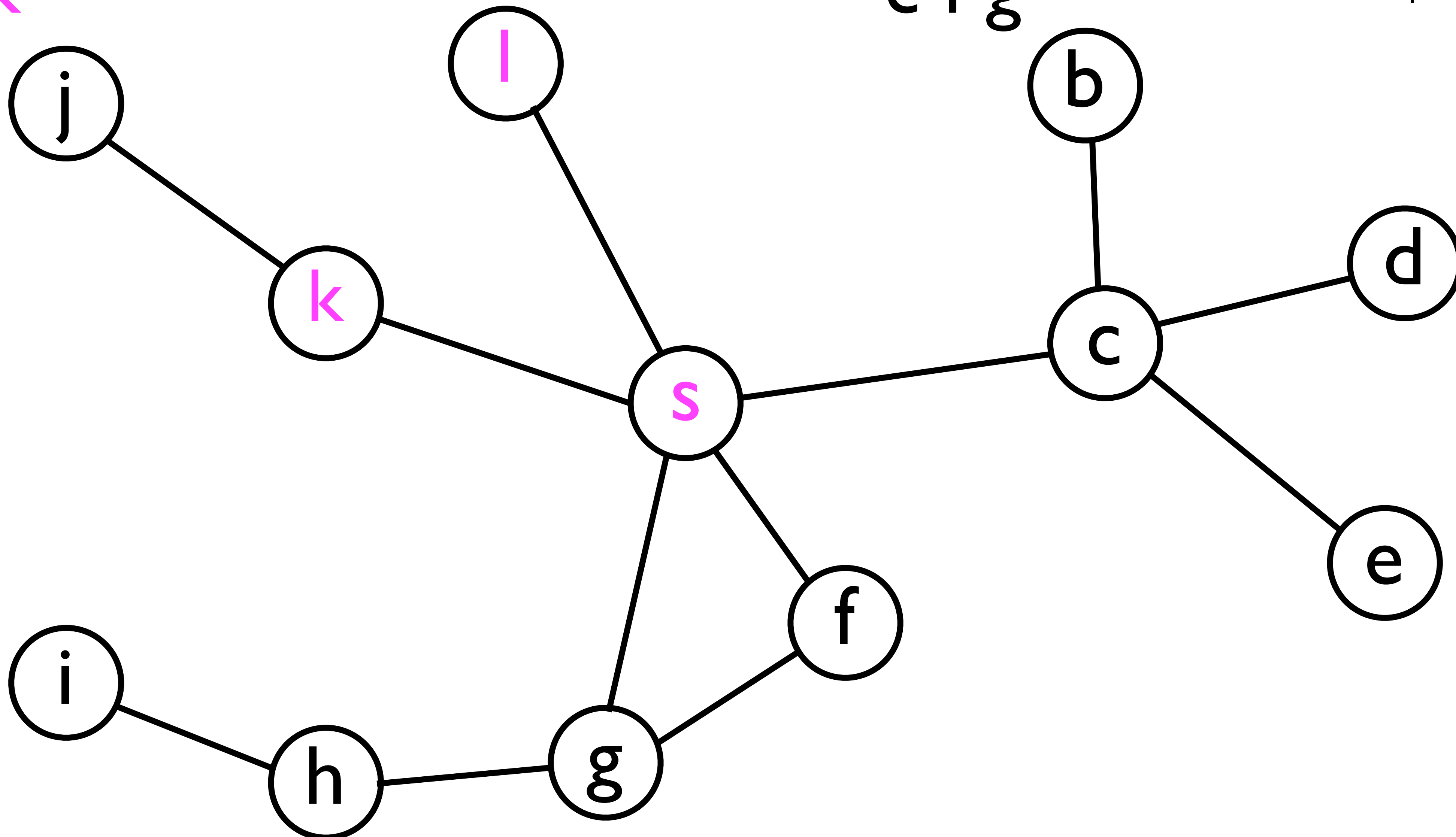
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s l k

`nodesToVisit`:

bottom c f g top



DFS in practice

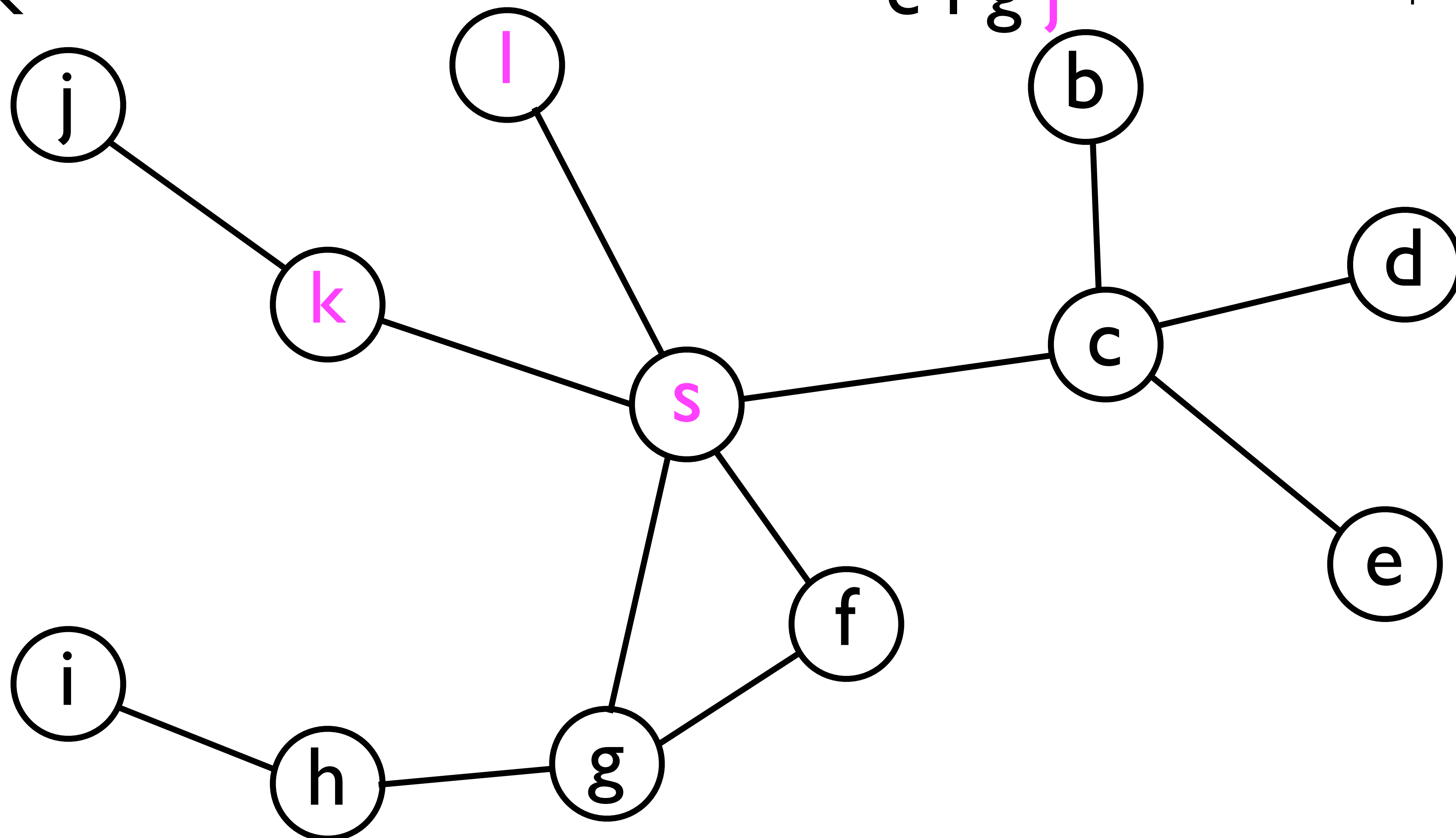
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s l k

`nodesToVisit`:

bottom c f g j top



DFS in practice

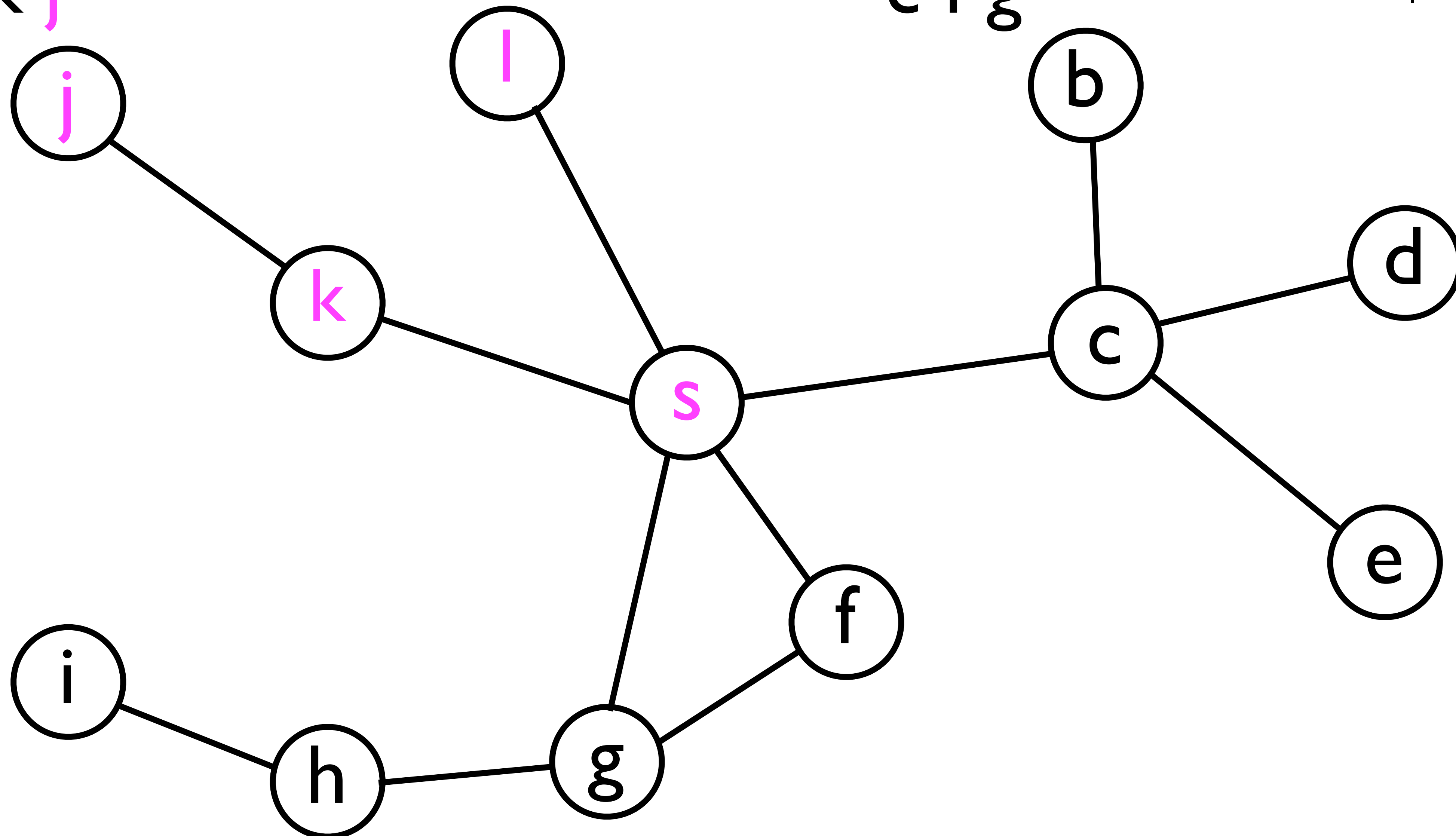
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s l k j

`nodesToVisit`:

bottom c f g top



DFS in practice

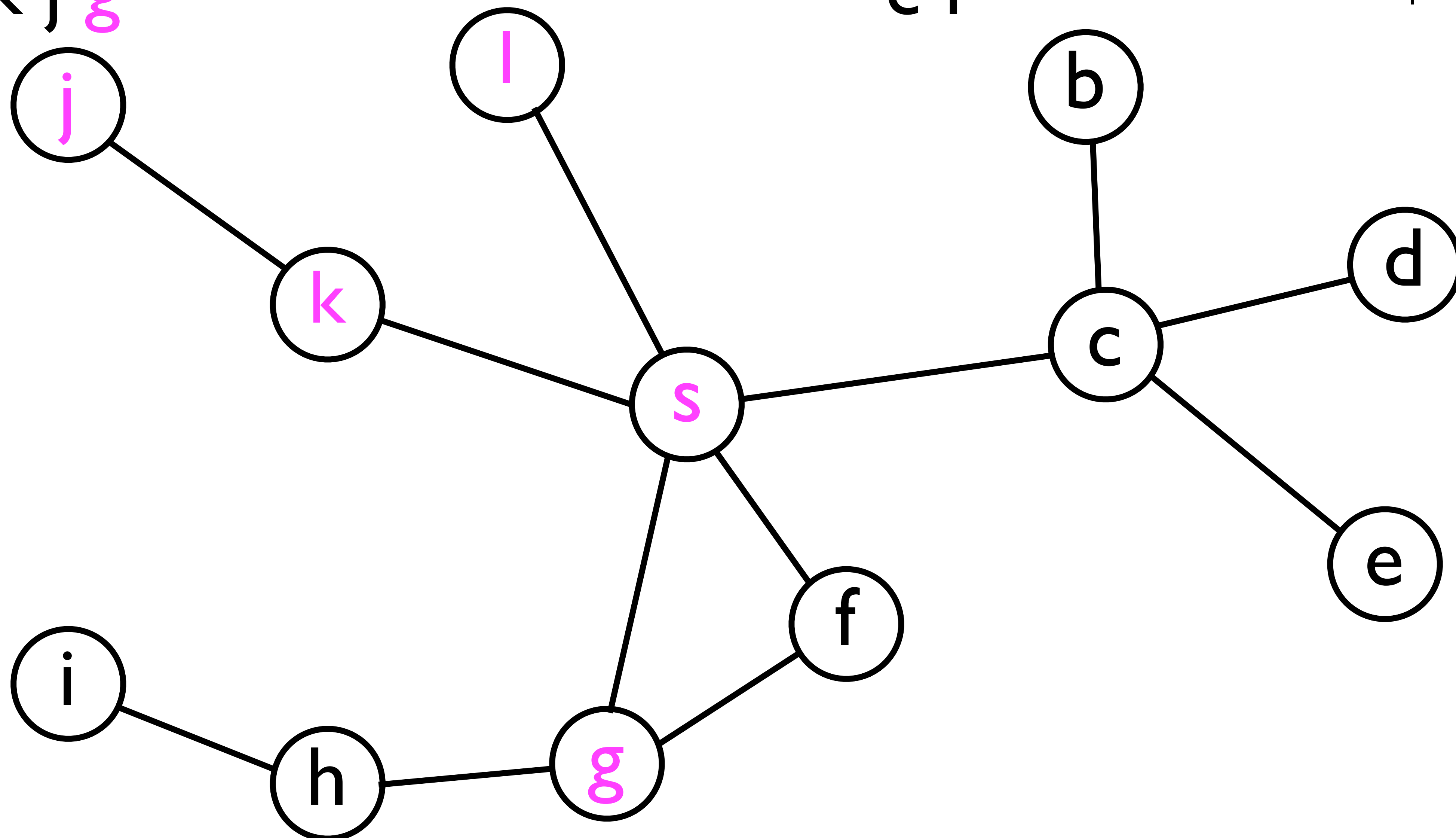
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s l k j g

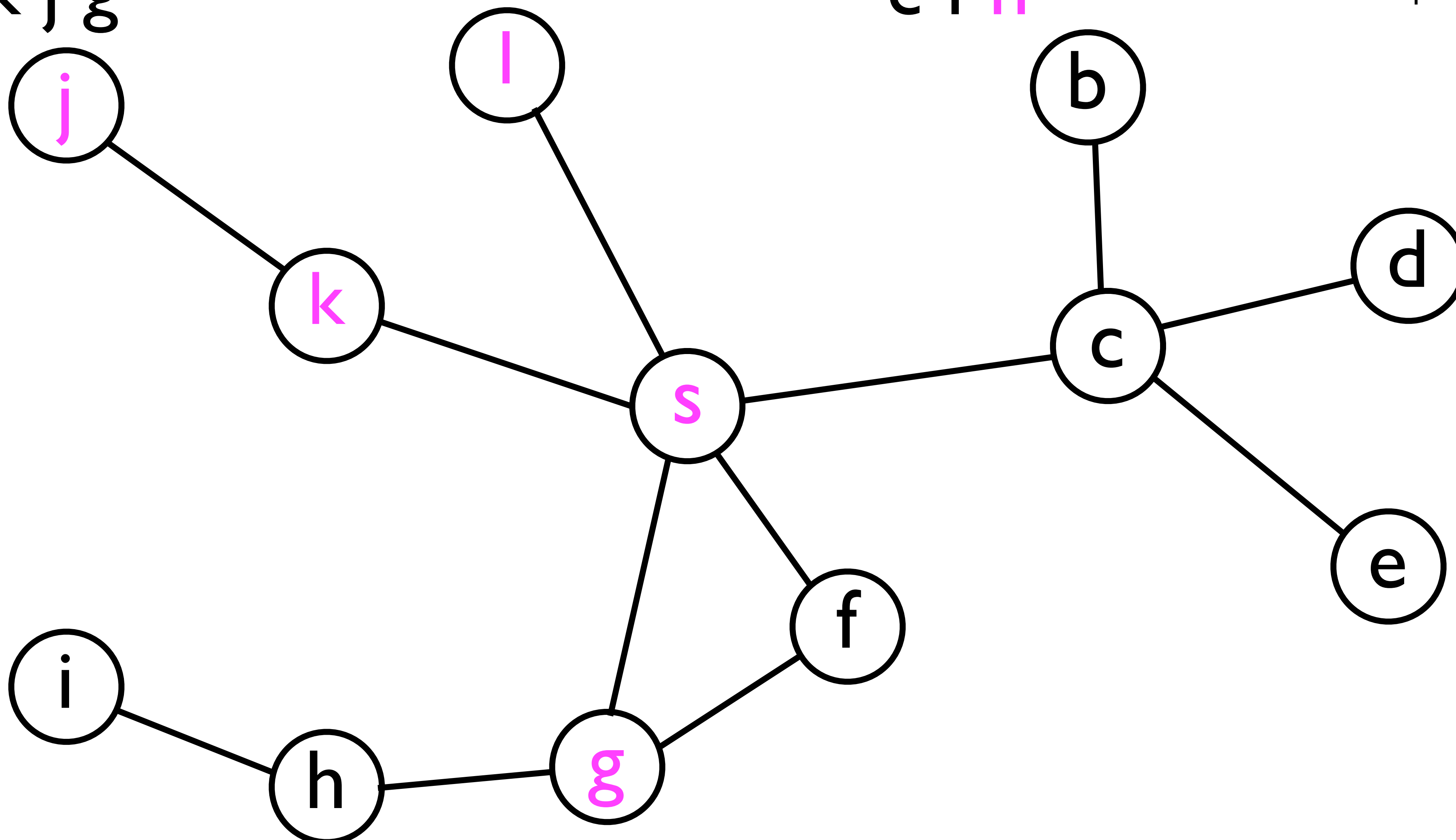
`nodesToVisit`:

bottom c f top



DFS in practice

- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...



DFS in practice

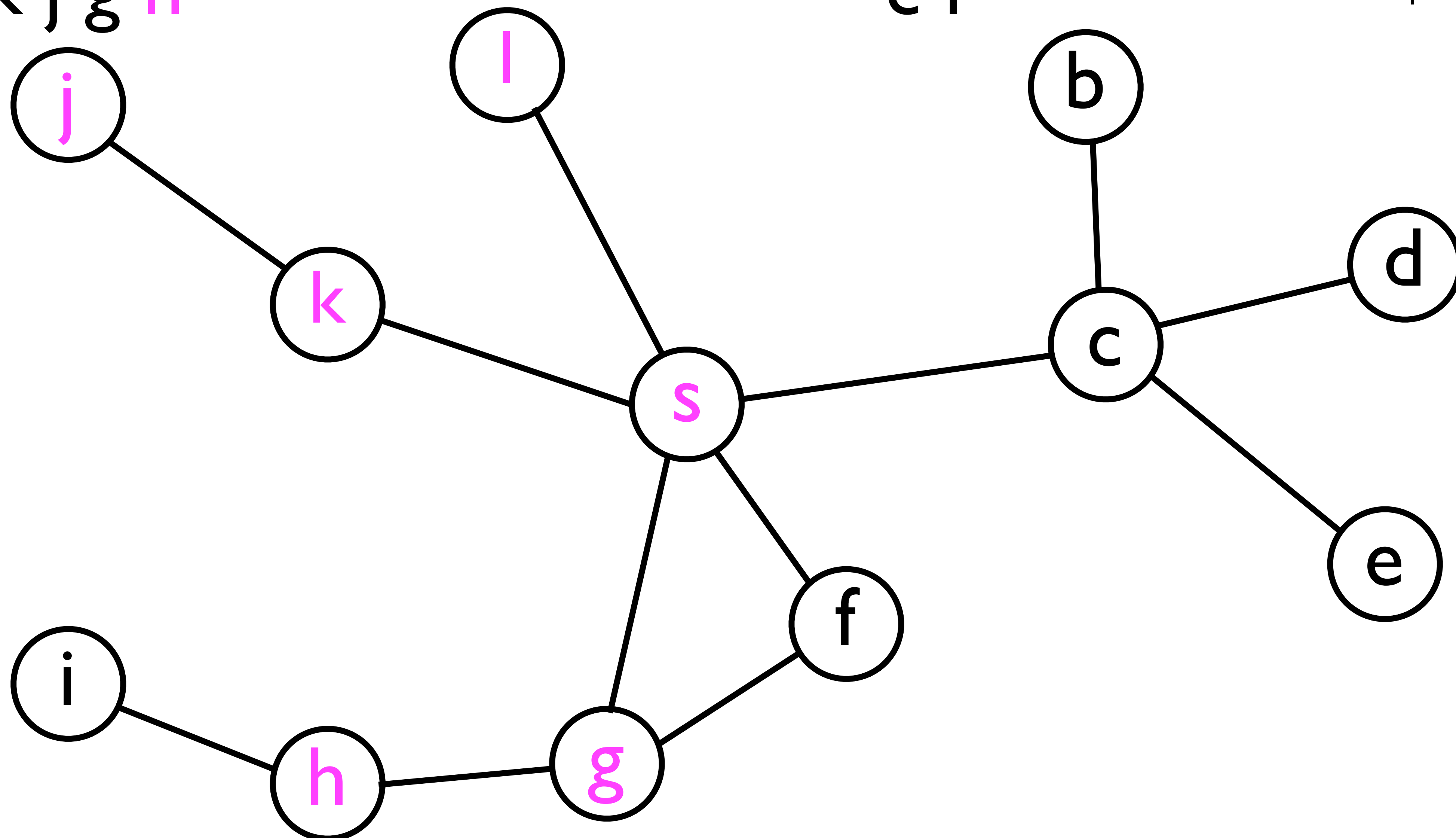
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s l k j g h

`nodesToVisit`:

bottom c f top



DFS in practice

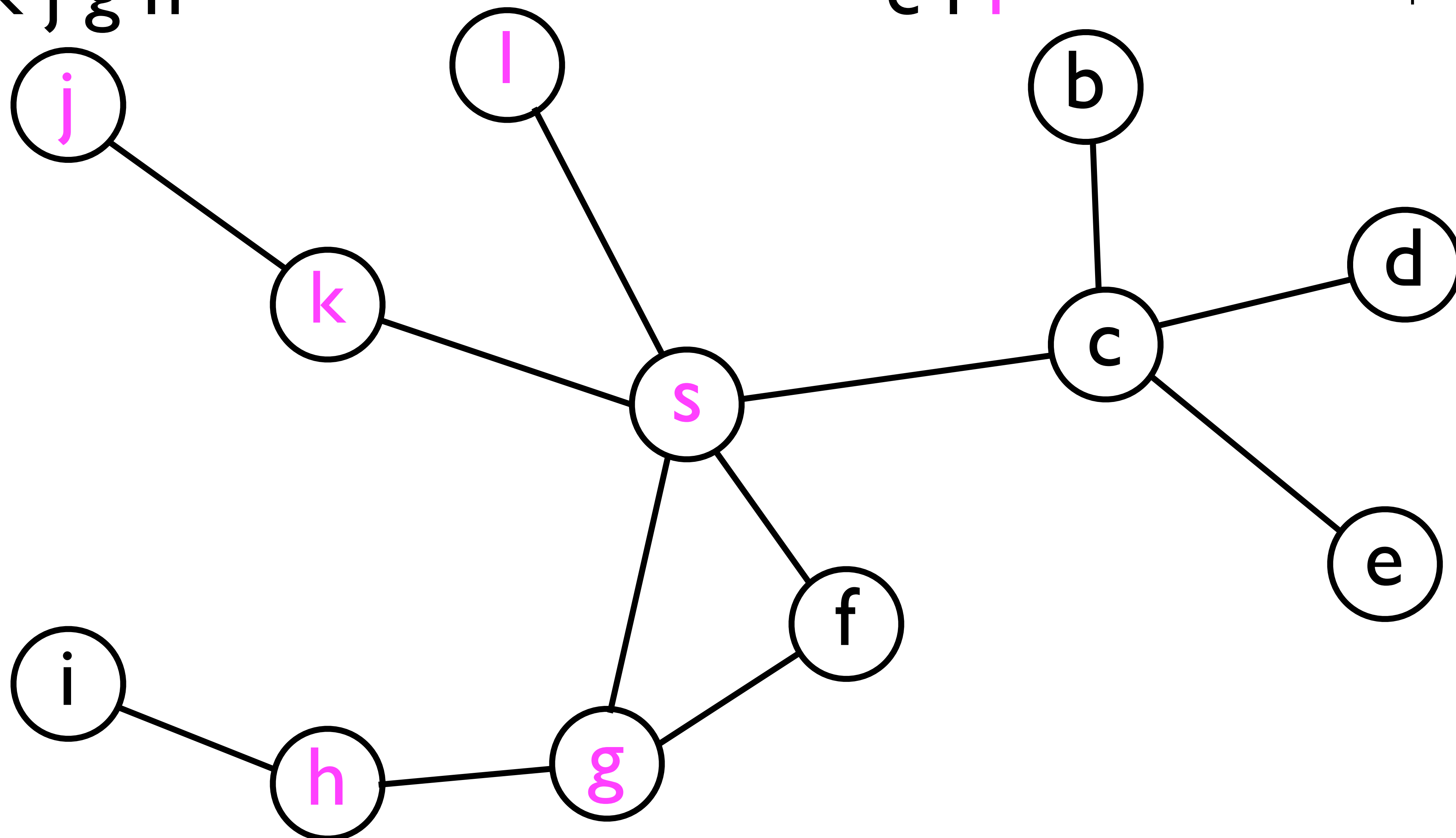
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s l k j g h

`nodesToVisit`:

bottom c f i top



DFS in practice

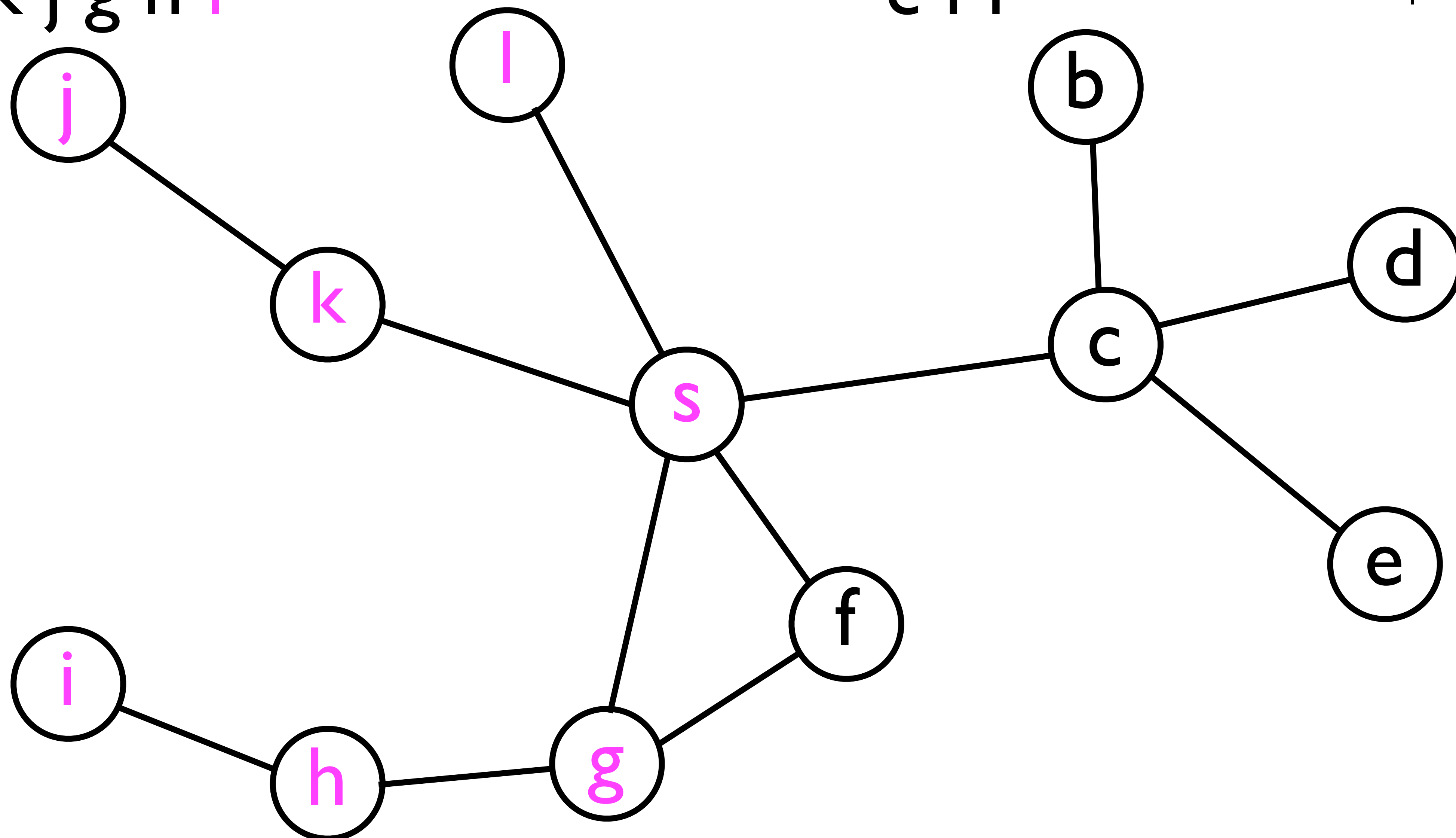
- Let's look at how the **`visitedNodes`** and **`nodesToVisit`** data structures are updated when exploring the graph from earlier...

`visitedNodes`:

s l k j g h i

`nodesToVisit`:

bottom c f i top



DFS in practice

- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

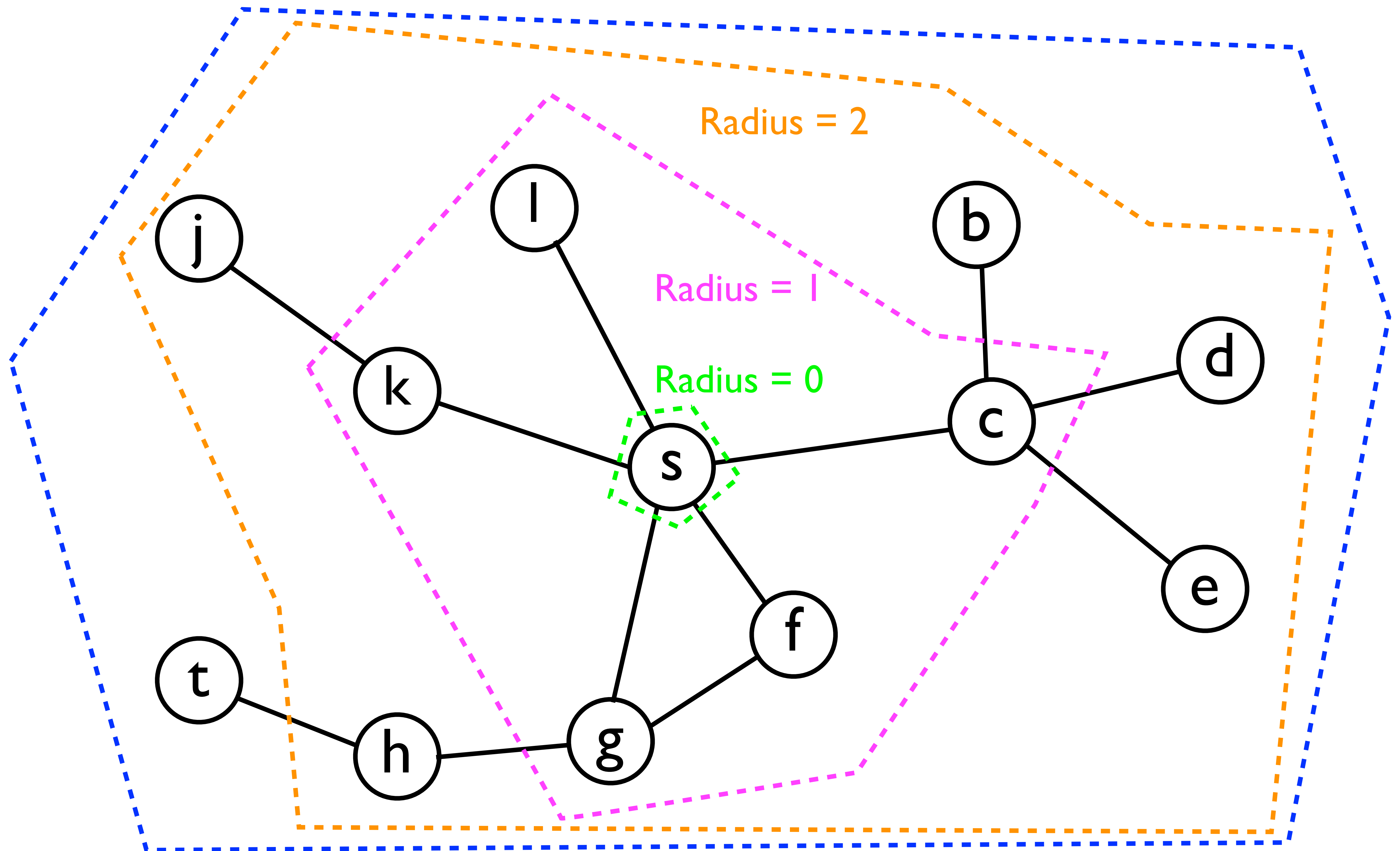
(and so on)

BFS and shortest paths

- It turns out that the BFS algorithm is also useful for finding shortest paths between two nodes in a graph.

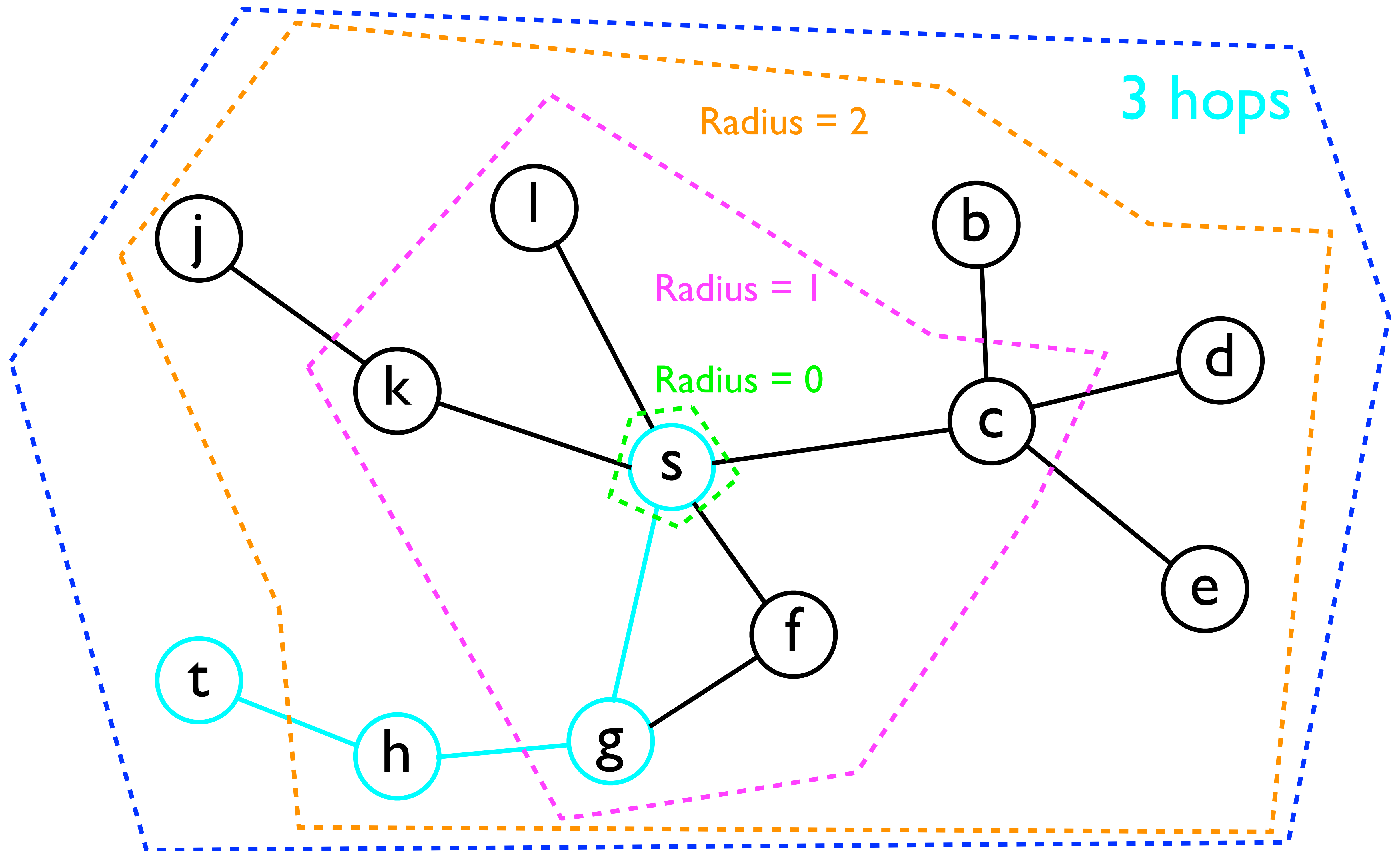
BFS and shortest paths

What is the *length* of shortest path from *s* to *t*?



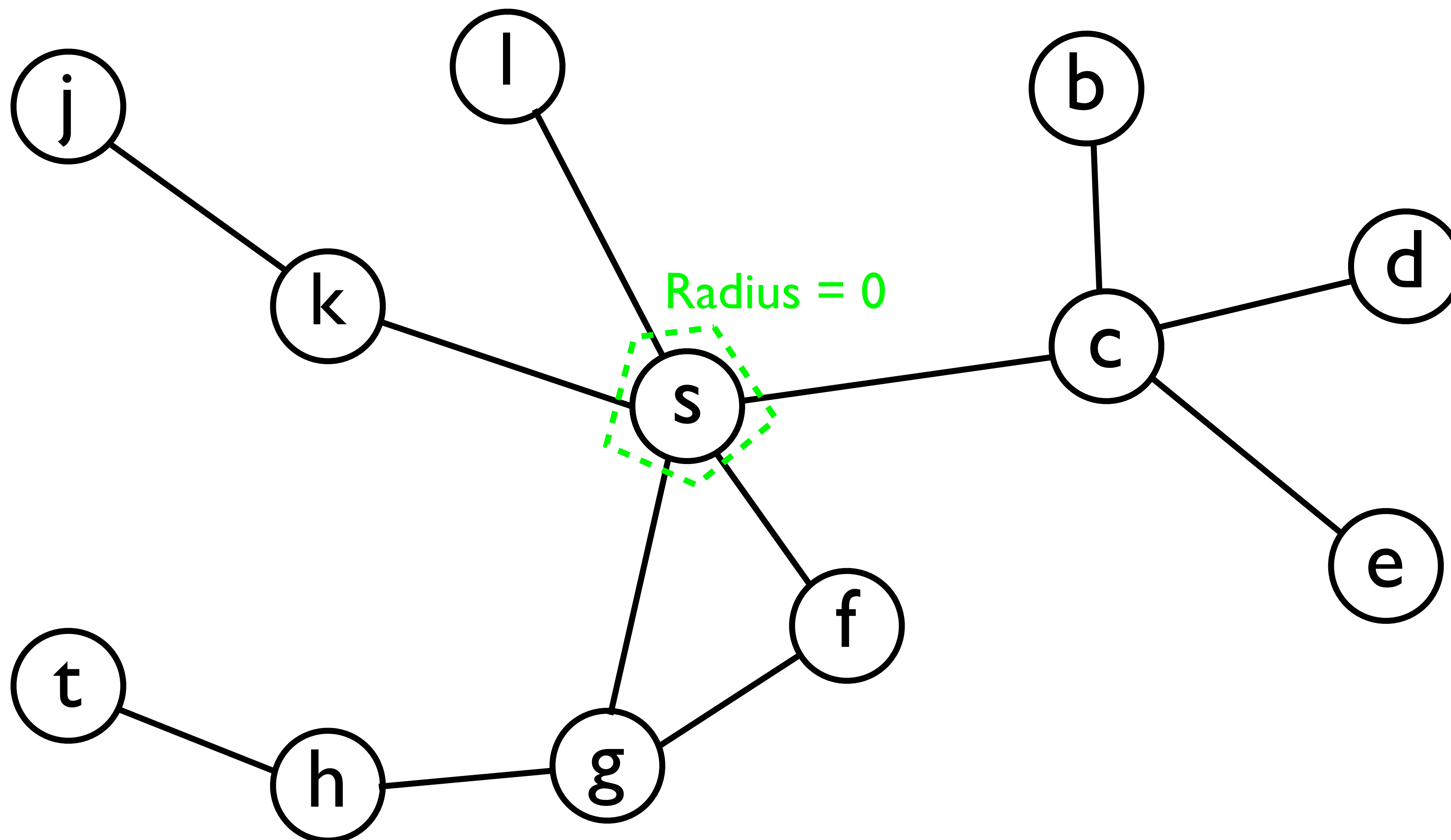
BFS and shortest paths

What is the *length* of shortest path from *s* to *t*?



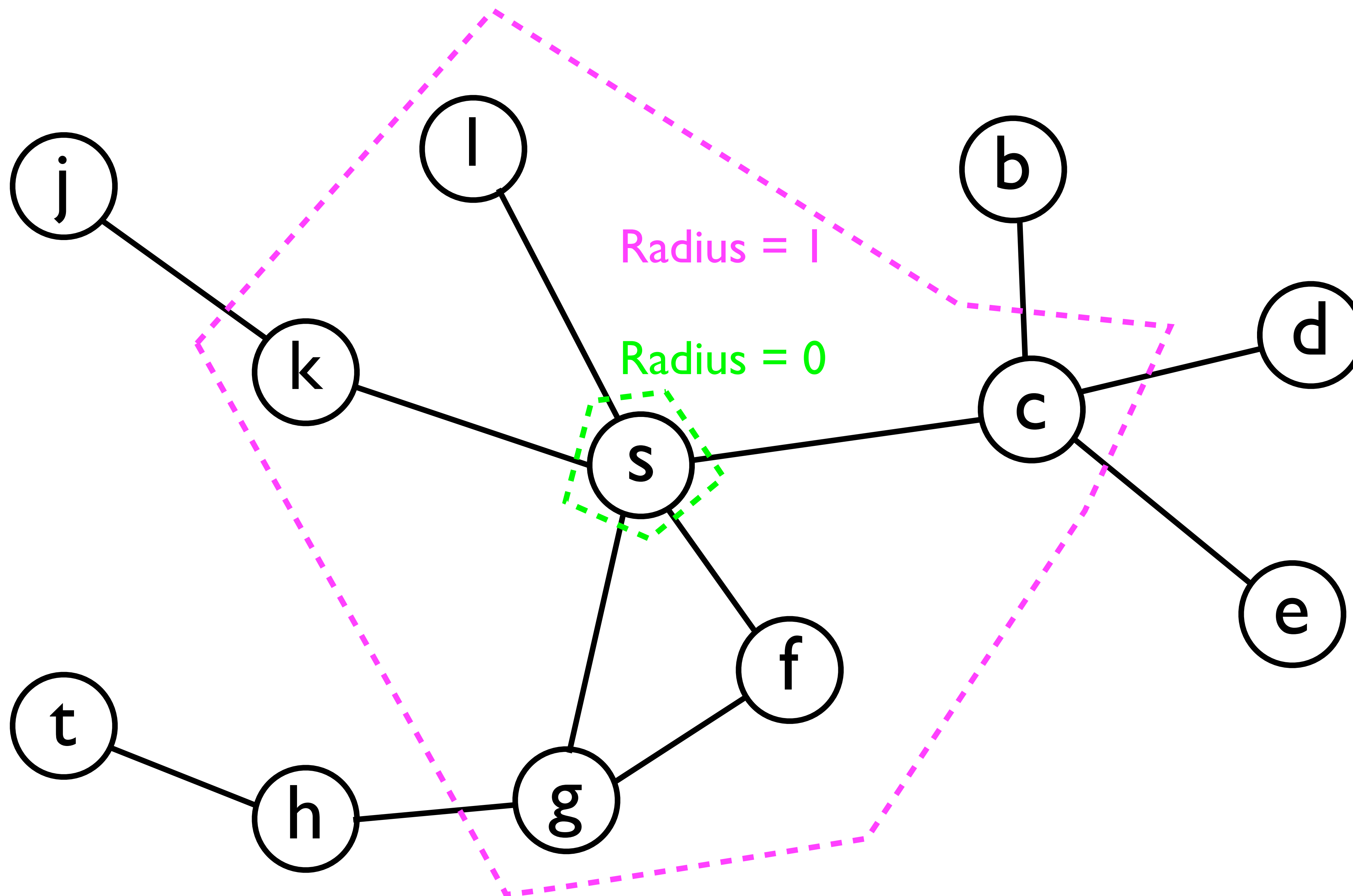
BFS and shortest paths

Are *s* and *t* within 0 hops apart? No.

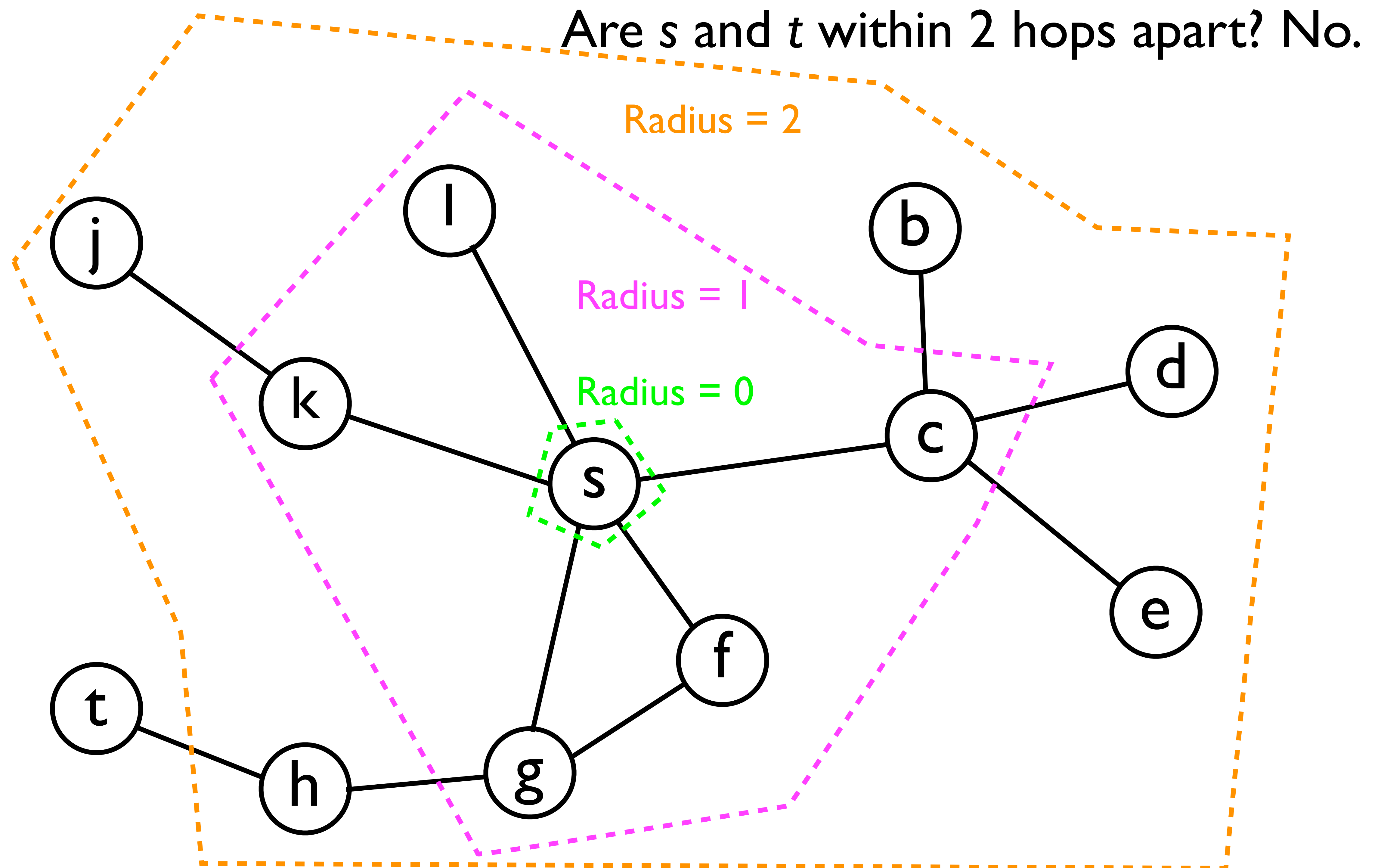


BFS and shortest paths

Are *s* and *t* within 1 hops apart? No.



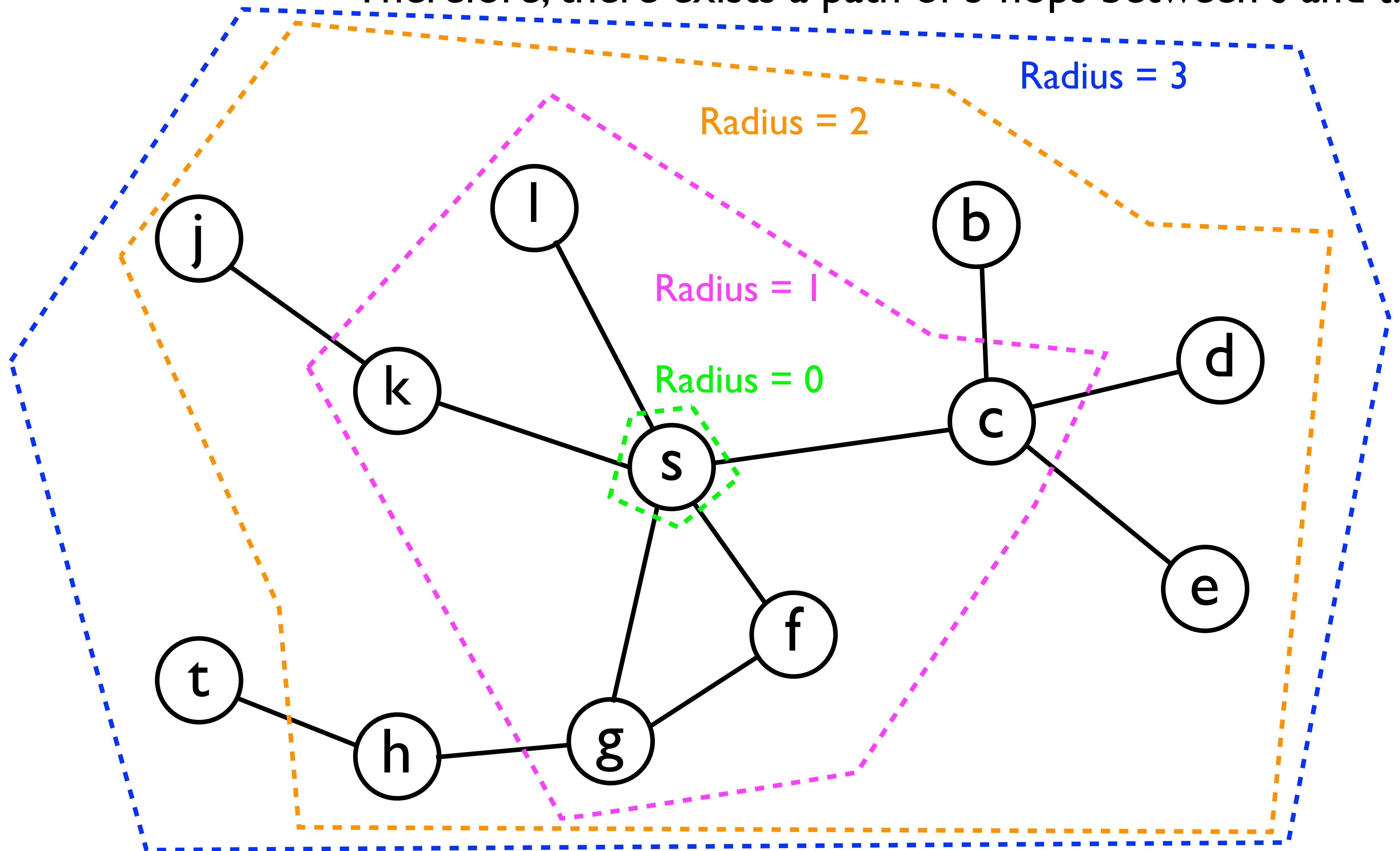
BFS and shortest paths



BFS and shortest paths

Are *s* and *t* within 3 hops apart? Yes!

Therefore, there exists a path of 3 hops between *s* and *t*.



Kevin Bacon Number demo.

Sorting

Facesbook demo

Sorting

- Given a data structure to store the user's data, one of the fundamental *operations* we may want to perform is to *sort* the data.
- Some ADTs already utilize the order relations among data to store data more efficiently.
  - Heaps always store the *largest* element at the top.
  - Binary search trees impose require data in *left* sub-tree be smaller than in *right* sub-tree.
- However, in this section we are interested in using the order relations to sort data stored in a *standard array*:
  - Fixed size
  - $O(1)$ time read/write access to any element.

Sorting

- Whole books have been written on sorting algorithms.
- Here, we present 6 of the most prominent sorting algorithms and discuss their relative merits.
- (Most of) the sorting algorithms we discuss are all based on *comparing and swapping elements*, i.e.:
  - To sort an `int[] array`, the only operations we perform are to *compare* `array[i]` and `array[j]` (for some *i* and *j*) and to possibly *swap* those elements.

Sorting

- It turns out that there is a *provable lower bound* on the time cost of any comparison-based sorting algorithm:
  - A lower bound on the worst-case performance of any comparison-based sort is O(n log n).
- Comparison-based sorts are the most generally applicable algorithms.
- However, if additional properties of the data can be assumed, then we do better with other non-comparison sorting procedures, such as bucket sort, radix sort, counting sort, etc.

Sorting

- Before discussing the individual sorting algorithms, let's clarify the setting:
- For concreteness, we will always sort an an array of *integers*.
  - In reality, it doesn't matter what type `T` they are as long as long `T` defines an *order relation* that is:
    - **Transitive**: if $x \le y$ and $y \le z$, then $x \le z$.
    - **Anti-symmetric**: if $x \le y$ and $y \le x$, then $x = y$.

Sorting

- If all the *n* data in our array are *unique*, then we can sort them either into *increasing* or *decreasing* order, i.e.:
  - **Increasing**: $e_1 < e_2 < e_3 < ... < e_{n-1} < e_n$
  - **Decreasing**: $e_1 > e_2 > e_3 > ... > e_{n-1} > e_n$
- If the array contains *duplicates*, then we can only achieve *non-decreasing* or *non-increasing* order:
  - **Non-decreasing**: $e_1 \le e_2 \le e_3 \le ... \le e_{n-1} \le e_n$
  - **Non-increasing**: $e_1 \ge e_2 \ge e_3 \ge ... \ge e_{n-1} \ge e_n$

Sorting properties

- There are several properties of sorting algorithms that we are interested in:

1. Asymptotic time costs in best, average, and worst cases.
2. Whether the algorithm can sort *in-place*:
    - An **in-place** sorting algorithms requires only $O(1)$ space outside of the array itself.
    - Non-in-place algorithms may have to copy the array into a temporary $O(n)$ buffer.

Sorting properties

3. Whether the sorting procedure is *stable*:
   - A **stable** sorting algorithm will maintain the relative order of *duplicate elements* in the sorted array compared to the input array, e.g.:
     - Unsorted array: 5_1 3 2 5_2 7
     - Sorted array (stable): 2 3 5_1 5_2 7
     - Sorted array (non-stable): 2 3 5_2 5_1 7

Sorting properties

- Stable sort -- why do we care?
  - Sometimes we may sort the *same* data on *different attributes* in sequence, e.g.:
    - *First* sort data based on a person's *age*.
      - **`p1`** > **`p2`** iff **`p1.age`** > **`p2.age`**
    - *Then* sort the *same* data based on a person's *country-of-residence*.
      - **`p1`** > **`p2`** iff **`p1.country`** > **`p2.country`**
  - We don't want the second sort to "mess up" the order of the first sort.

Sorting properties

- Example -- unsorted data:

  Richard, age 47, USA
  Ronald, age 16, Paraguay
  Gerald, age 32, Monaco
  Jimmy, age 97, Indonesia
  George, age 18, USA
  Bill, age 54, Monaco
  Ganymede, age 88, USA

Sorting properties

- Example -- data sorted by age.

  Ronald, age 16, Paraguay
  George, age 18, USA
  Gerald, age 32, Monaco
  Richard, age 47, USA
  Bill, age 54, Monaco
  Ganymede, age 88, USA
  Jimmy, age 97, Indonesia

Sorting properties

- Example -- data sorted by country (stable).

Jimmy, age 97, Indonesia
Gerald, age 32, Monaco
Bill, age 54, Monaco
Ronald, age 16, Paraguay
George, age 18, USA
Richard, age 47, USA
Ganymede, age 88, USA

Within each country, the data are still sorted by age.

Sorting properties

- Example -- data sorted by country (non-stable).

Jimmy, age 97, Indonesia
Bill, age 54, Monaco
Gerald, age 32, Monaco
Ronald, age 16, Paraguay
Richard, age 47, USA
George, age 18, USA
Ganymede, age 88, USA

Within each country, the data are *not* sorted by age.

Sorting properties

3. The time cost of the algorithm on an array that is *already sorted*.

    - Oftentimes, we will sort an array that *might* already be sorted.
    - Some algorithms perform better or worse depending on whether the input array is already sorted or perhaps "almost" sorted.

Bogosort.

Bogosort

- Bogosort is a probabilistic sorting algorithm that *randomly shuffles* the input array until it is *sorted*:
    - While array is not sorted:
        - Randomly permute the contents of array.
        - Check if the array is sorted.

Bogosort

- Bogosort is a probabilistic search algorithm that *randomly shuffles* the input array until it is *sorted*:
  - While array is not sorted:
    - Randomly permute the contents of array.
    - Check if the array is sorted.
- Is this algorithm guaranteed to finish?
  - Yes, given infinite time.
  - Given any finite number of iterations, it is always possible that Bogosort hasn't "found" the right permutation yet.

Bogosort

- The (average-case) time costs of Bogosort are, as a whole, rather "unsatisfying":
  - While array is not sorted: $O(n!)$ (expected)
    - Randomly permute the contents of array. $O(n)$
    - Check if the array is sorted. $O(n)$
- Hence, in the *average* case, Bogosort takes time $O(n!) = O(n^n)$.
- In *best* case, Bogosort takes $O(n)$ time to permute the array once and check that the permuted array is sorted.
- *Worst* case: undefined -- may never terminate in finite time.

Bogosort

- Does there exist some sorting algorithm which gives us asymptotically performance than Bogosort?
  - Hard to say.

Approach 1: sorted- and unsorted- parts

- Two prominent algorithms (with time costs better than $O(n^n)$) that we examine are *selection sort* and *insertion sort.*
- Both these algorithms work by partitioning the input array into:
  - A **sorted part**: a sub-array that is already sorted.
  - An **unsorted part**: a sub-array that has not yet been processed and which might not yet be sorted.

Approach 1: sorted- and unsorted- parts

- For example, consider the following array:

6 1 3 2 4 5 7 8

Unsorted part　　　　　Sorted part

- The algorithms differ in how these parts are created and maintained.

Selection sort.

Selection sort

- The main idea of selection sort is to repeatedly (*n* times) *select* the largest element in the unsorted part, and *move* it into the sorted part.
- If the unsorted part contains *n* elements, then we can find the largest element in *n* operations.
  - Adding it to the sorted part just takes O(*1*) time.

Selection sort

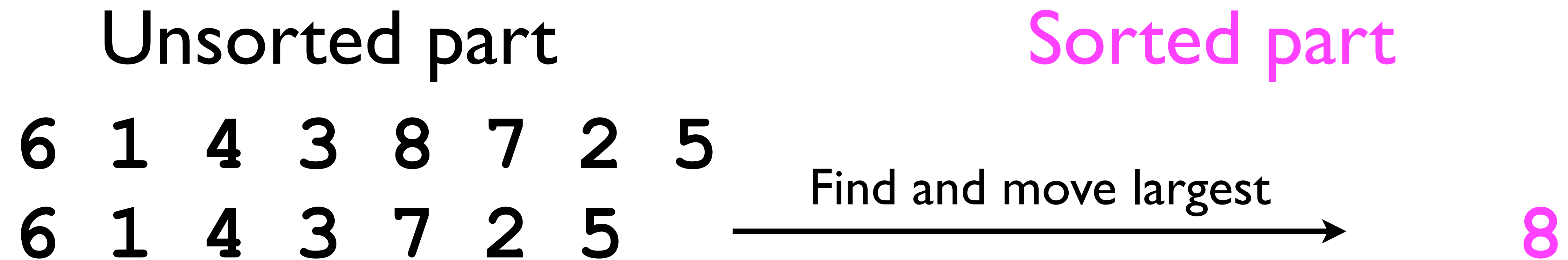
- Example:

Unsorted part | Sorted part

6 1 4 3 8 7 2 5

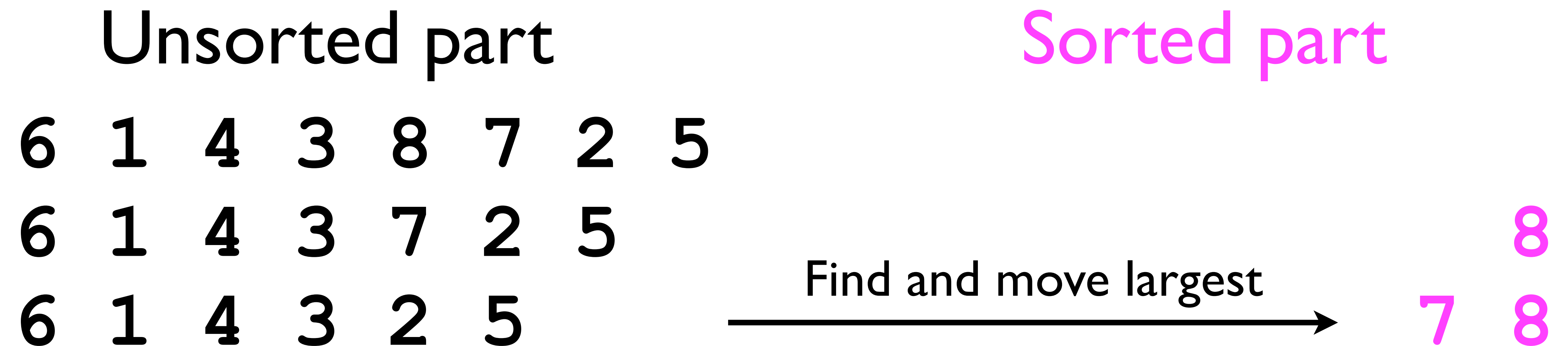
Selection sort

- Example:



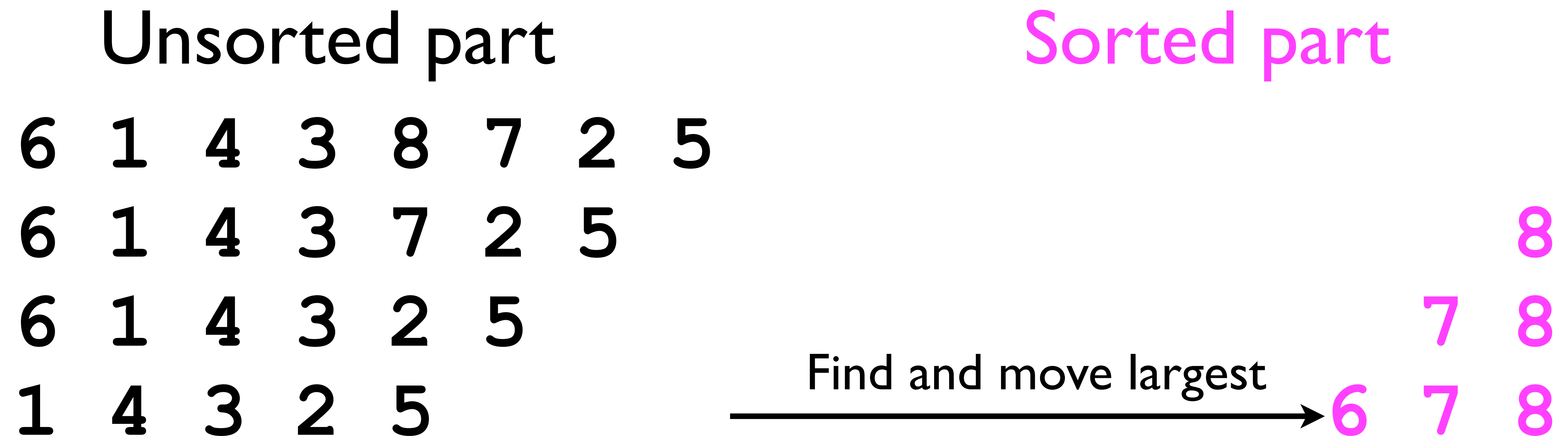
Selection sort

- Example:



Selection sort

- Example:



Selection sort

- Example:

Unsorted part	Sorted part
`6 1 4 3 8 7 2 5`	
`6 1 4 3 7 2 5`	`8`
`6 1 4 3 2 5`	`7 8`
`1 4 3 2 5`	`6 7 8`
`1 4 3 2`	`5 6 7 8`
`1 3 2`	`4 5 6 7 8`
`1 2`	`3 4 5 6 7 8`
`1`	`2 3 4 5 6 7 8`
Done.	`1 2 3 4 5 6 7 8`

Selection sort

- The figure above suggested that we maintain *two* separate arrays: one for the *unsorted part* (the *input array*), and one for the *sorted part*.
- However, we can make selection sort operate *in-place* if we *swap* the largest element in the unsorted part with the *right-most* element in the unsorted part...

Selection sort

- Example:

Unsorted part Sorted part

```
6 1 4 3 8 7 2 5
```

Selection sort

- Example:

Unsorted part Sorted part

6 1 4 3 5 7 2 8

Selection sort

- Example:

Unsorted part Sorted part

```
6 1 4 3 5 2 7 8
```

Selection sort

- Example:

Unsorted part Sorted part

2 1 4 3 5 6 7 8

Selection sort

- Example:

Unsorted part Sorted part

1 2 3 4 5 6 7 8

Selection sort

- Pseudocode:

```
void selectionSort (int[] array) {
  While size of unsorted part > 0:
    Find largest element e of unsorted part
    Swap e with right-most element of unsorted par
}
```

- The while loop iterates n times.
- Finding the largest element takes time n, n-1, n-2, ..., down to 1 depending on the particular loop iteration.
- Swapping takes $O(1)$ time.
- Total time cost:

  for swaps

  $n + (n-1) + (n-2) + ... + 2 + 1 + n*O(1) = n(n-1)/2 + O(n)$

  $= O(n^2)$

Selection sort

- Pseudocode:

```
void selectionSort (int[] array) {
  While size of unsorted part > 0:
    Find largest element e of unsorted part
    Swap e with right-most element of unsorted par
}
```

- Note that this time analysis applies to the worst, best, and average cases.
  - The number of operations does not vary with the input.
  - In particular, the if the input is already sorted, the algorithm still takes time $O(n^2)$.

Selection sort: stability

- Is selection sort *stable*, i.e., for any duplicate input elements in the input array, will the sorted array preserve their relative order?
    - It depends on the implementation.
- When finding the largest element in the unsorted part, if ≥ 2 elements are both maximal, then the selection sort may pick any of them to "move" to the sorted part.
    - If the algorithm chooses the *last* maximal element to move, then the sort is stable.

Selection sort: stability

- Example:

Unsorted part	Sorted part
5_1 2 5_2 1	
5_1 2 1	5_2
2 1	5_1 5_2
1	2 5_1 5_2
	1 2 5_1 5_2

If we move *last* instance of the largest element to the sorted part, then the search is stable.