CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Seventeen
30 Aug 2011

More on graphs.

Graphs

- Recall:

  - A graph $G = (N, E)$ consists of a set N of nodes and a set E of edges.

  - Graphs are typically used to represent *relationships* (edges) among *things* (nodes), e.g.:

    - Facebook users, mobile phone users, variables in a computer program.

  - A graph's *edges* can be represented using either an *adjacency matrix A* or an array of *adjacency lists* (one list for each node $n \in N$).

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*.

3, 1, and 4 are reachable from 2.

6 and 5 are not reachable from 2.
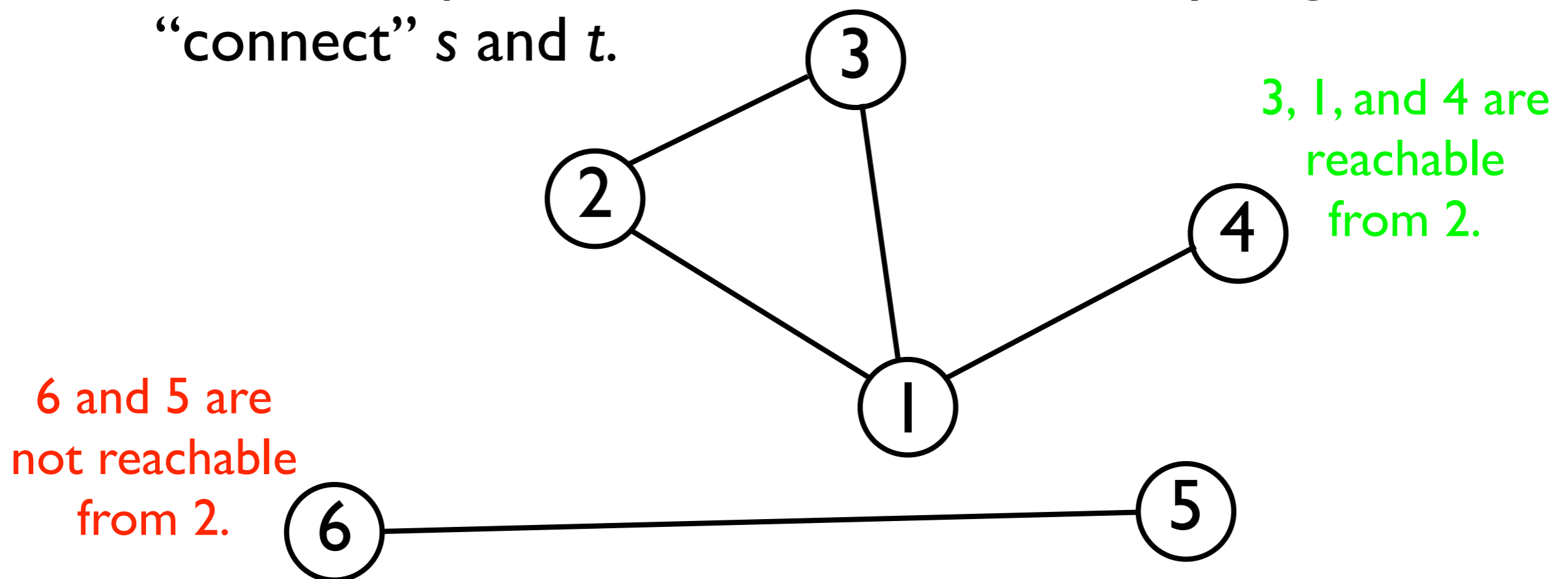
# Node discovery and shortest paths

- 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 "radially outwards" from *s*:

    - First, search for all nodes 1 step away from *s*.

    - Next, search for all nodes 2 steps away from *s*.
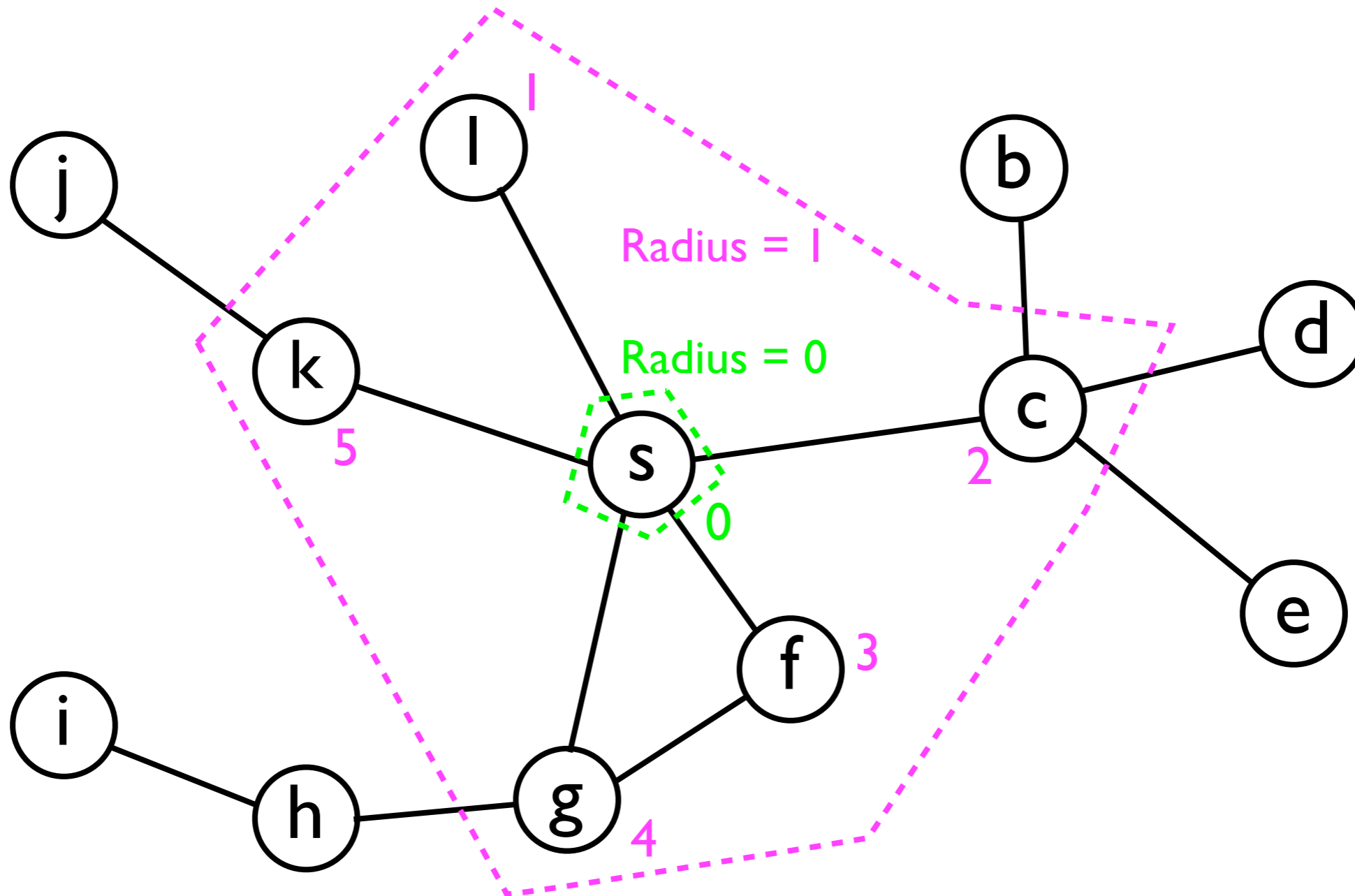
    - Then, search for all nodes 3 steps away ...

BFS

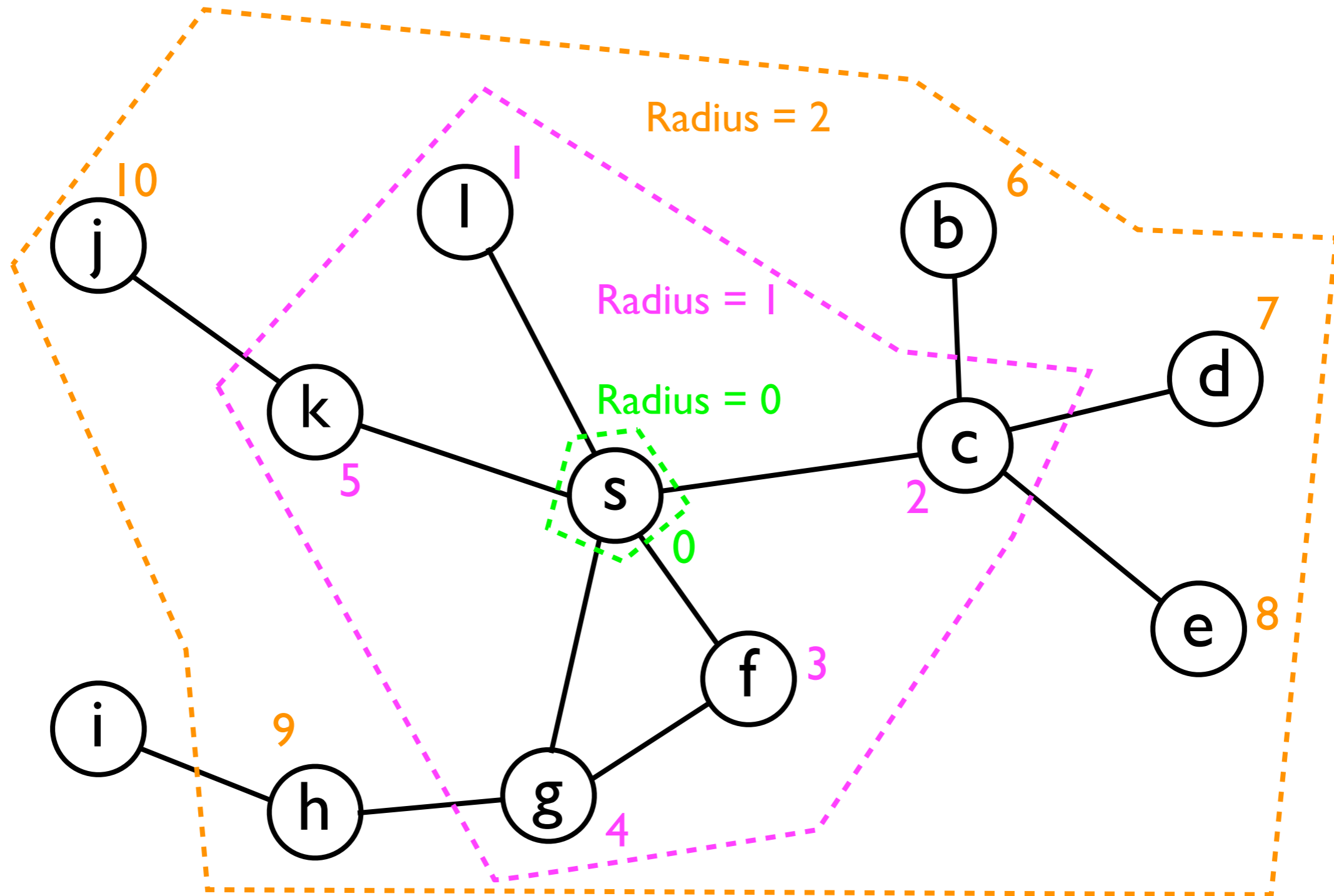A BFS starts with at a node *s* and "radiates outwards".
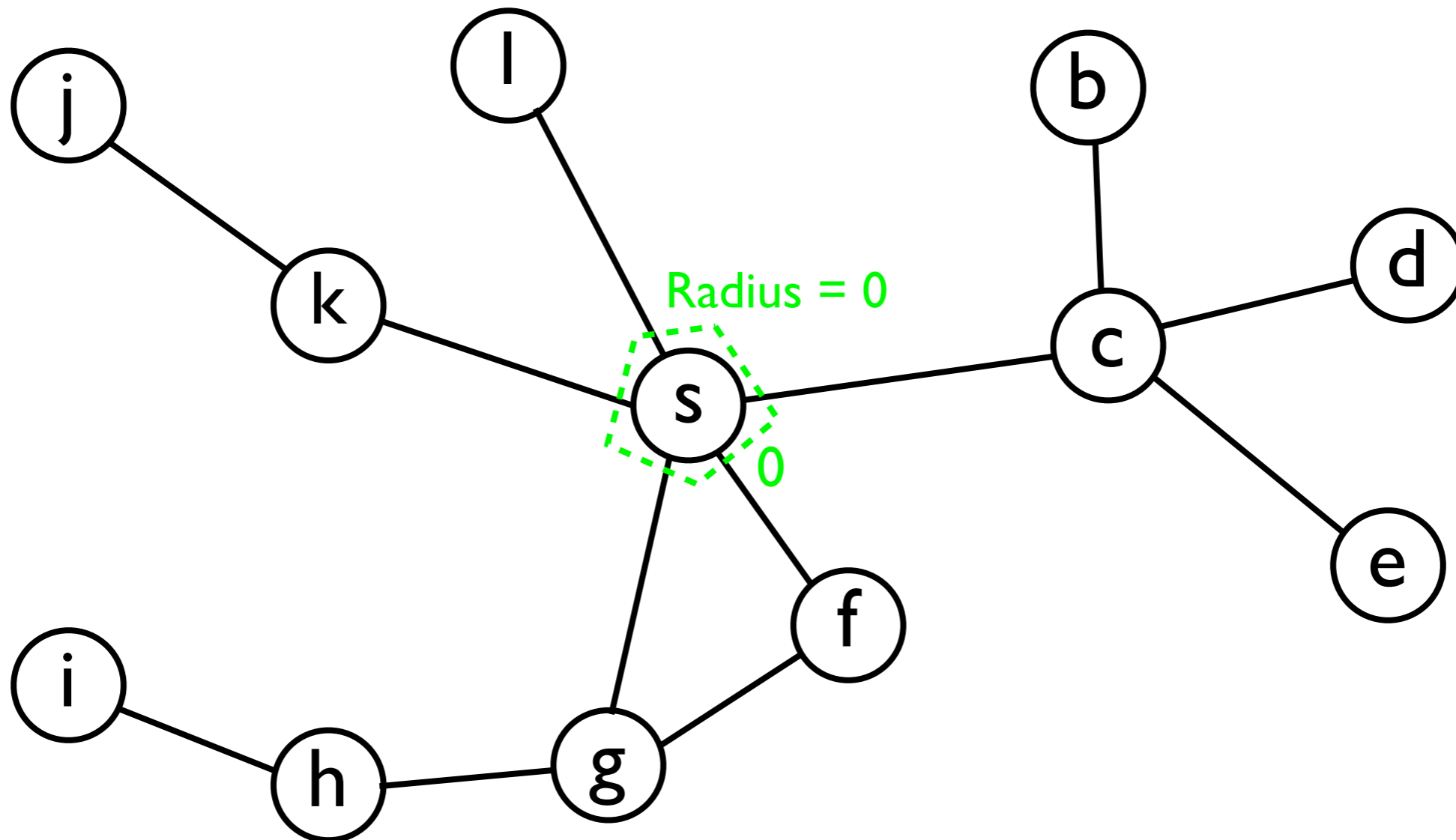
BFS

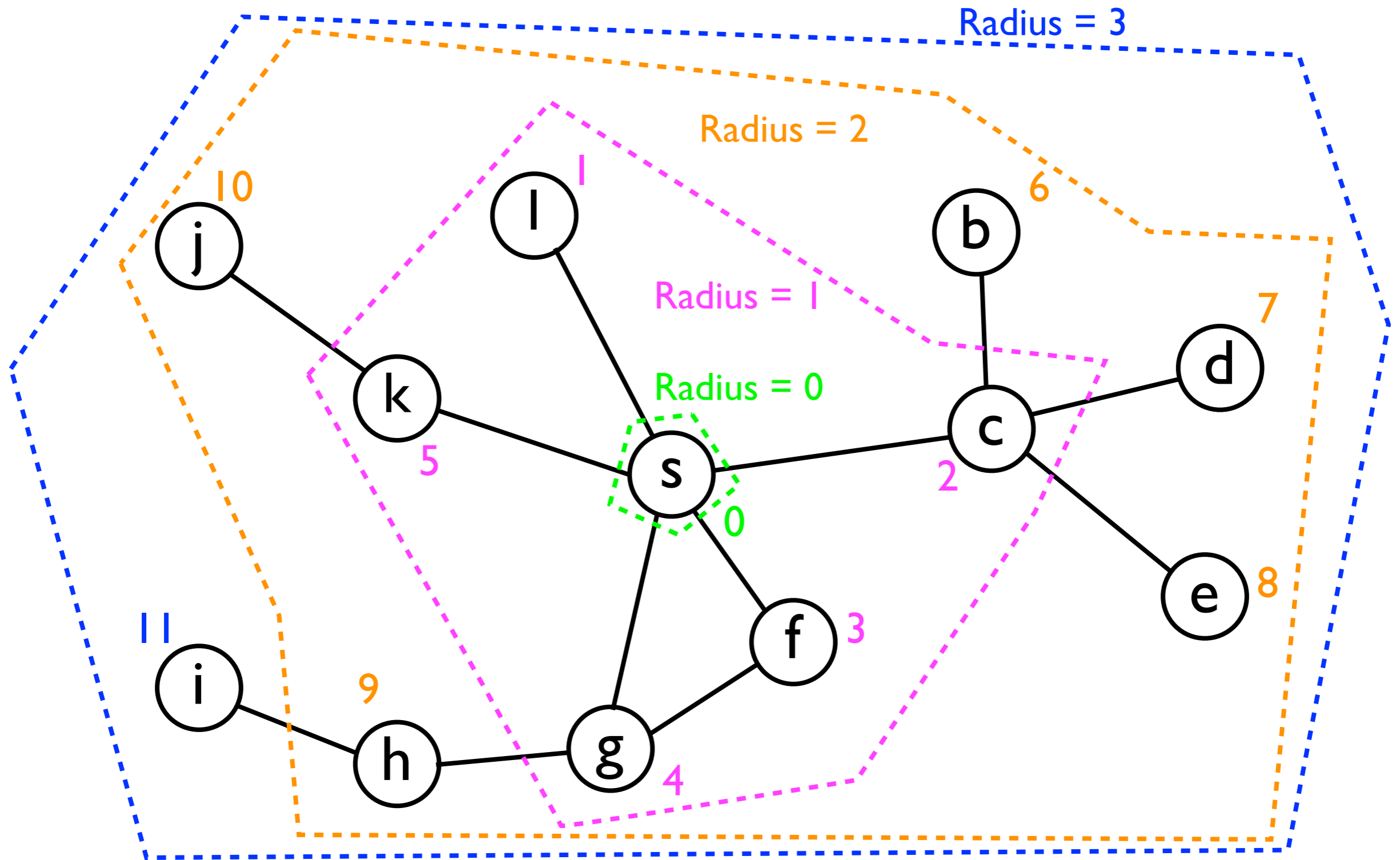A BFS starts with at a node *s* and "radiates outwards".

BFS

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

BFS

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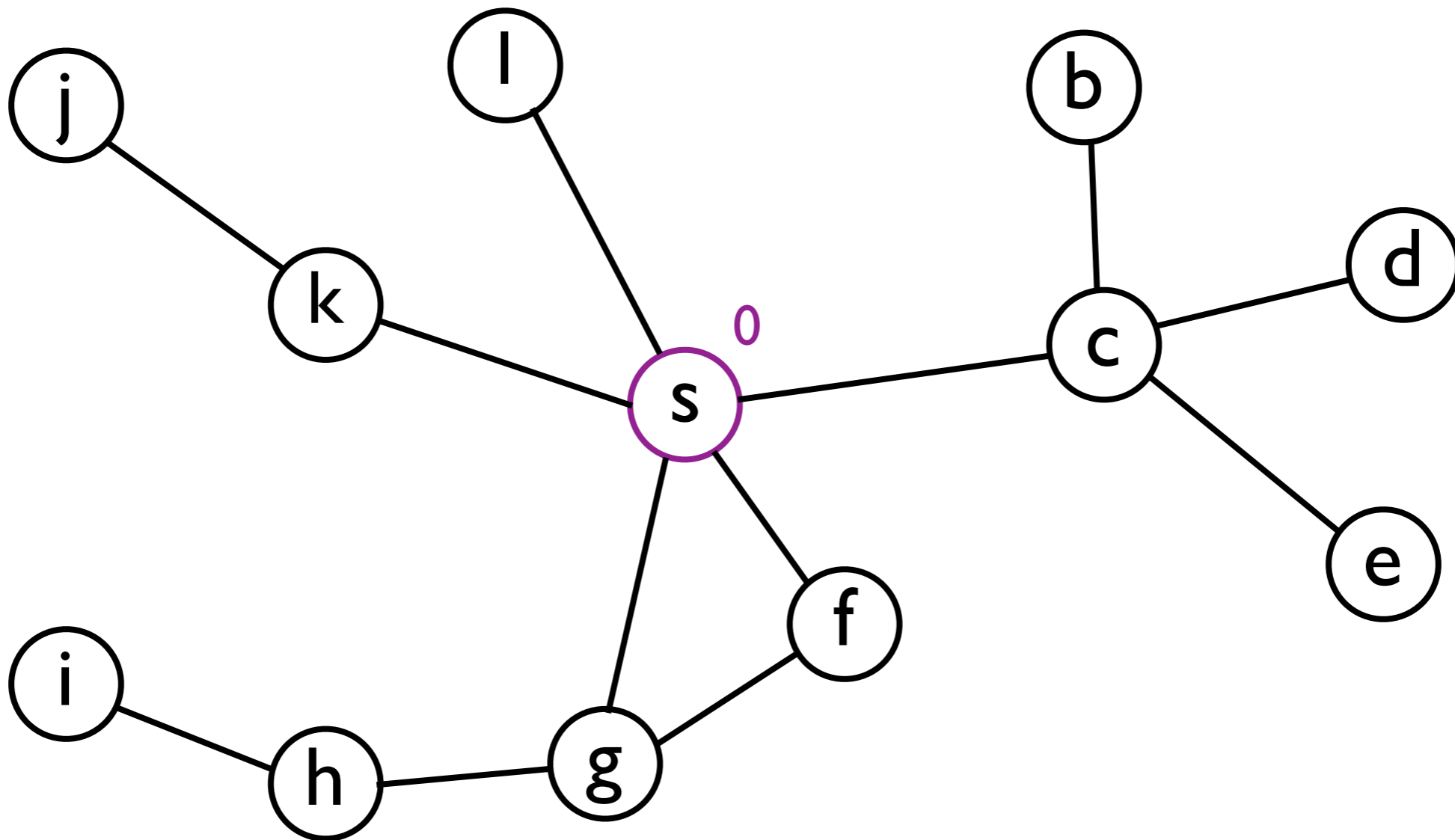
BFS and DFS

- 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".
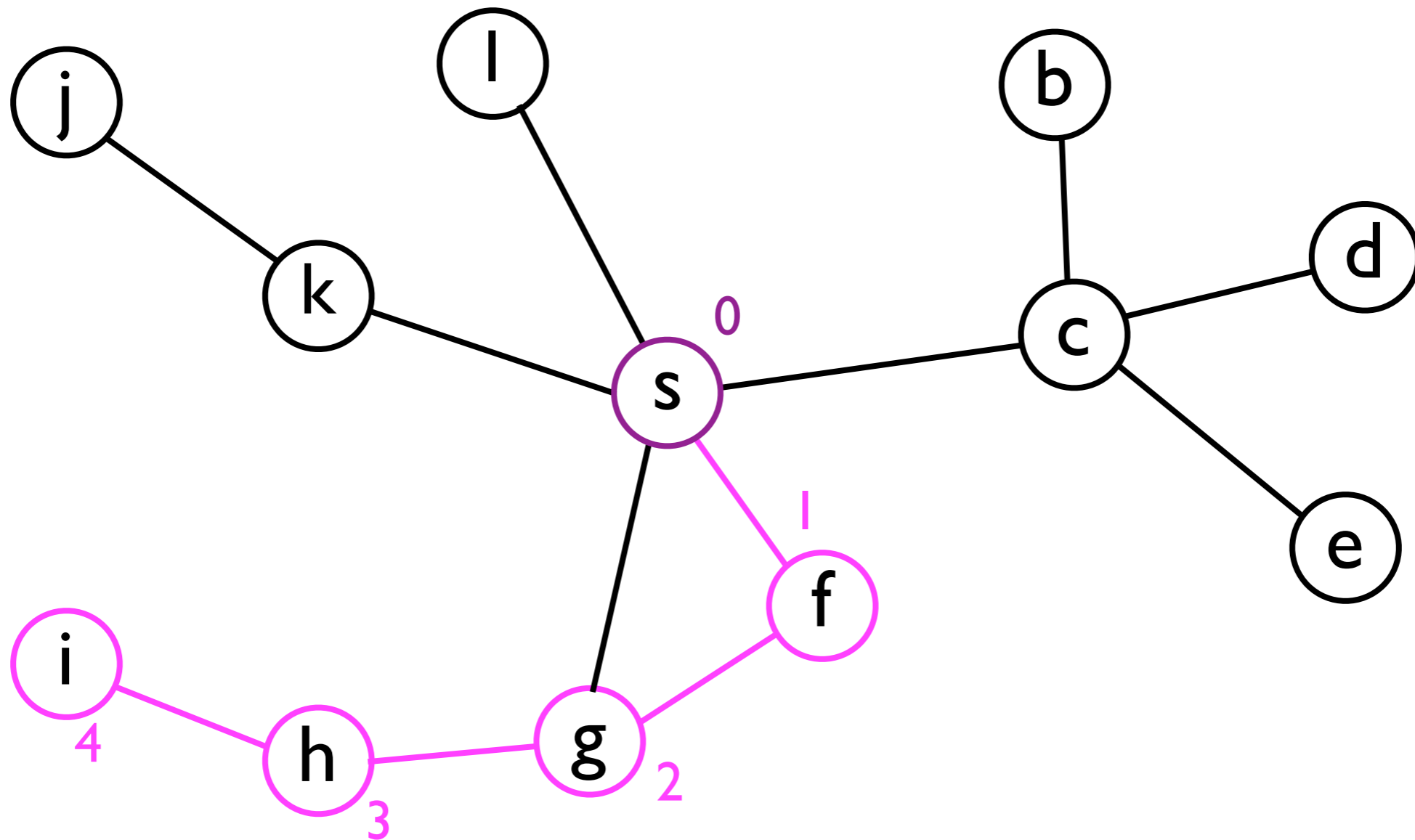
DFS

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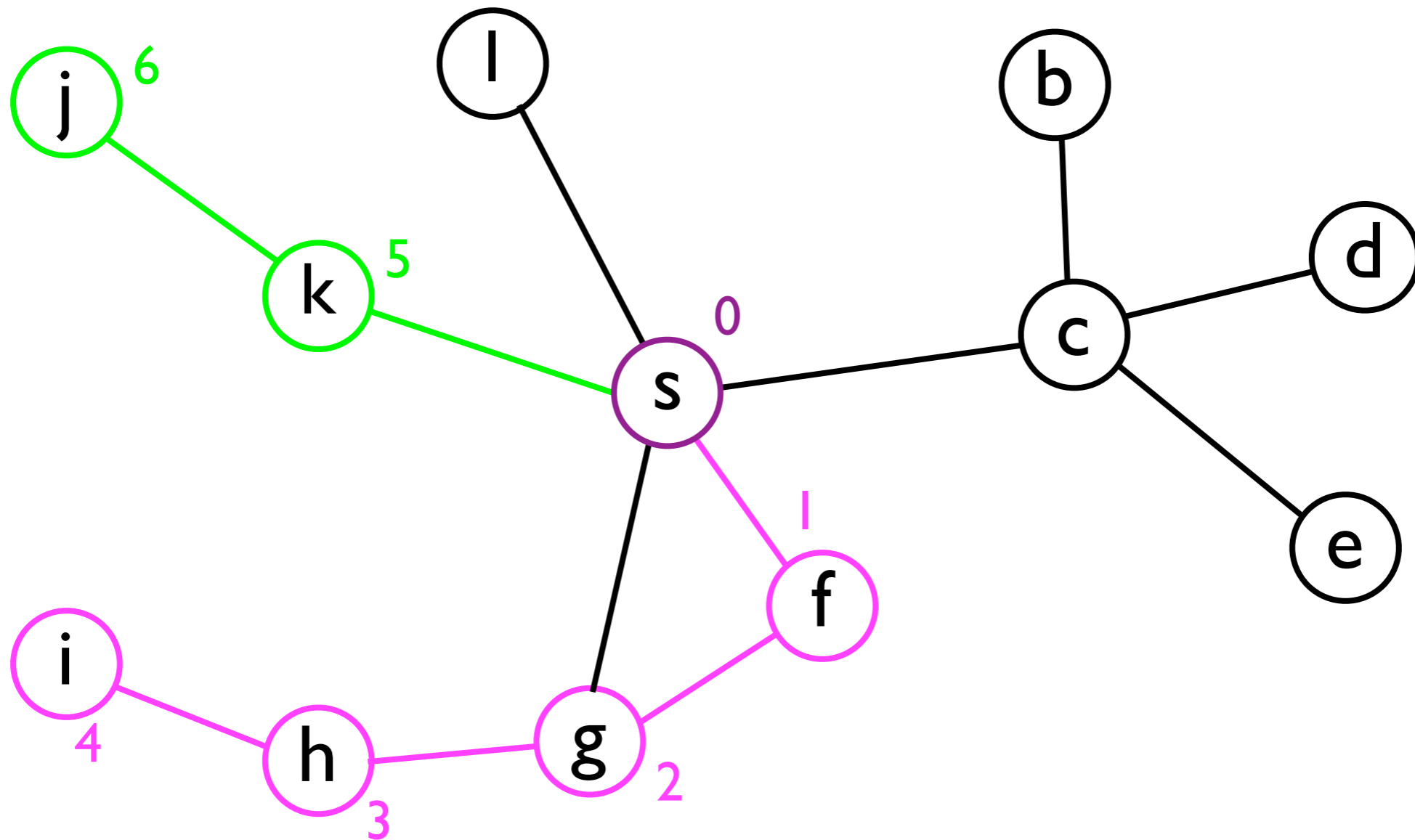
DFS

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".
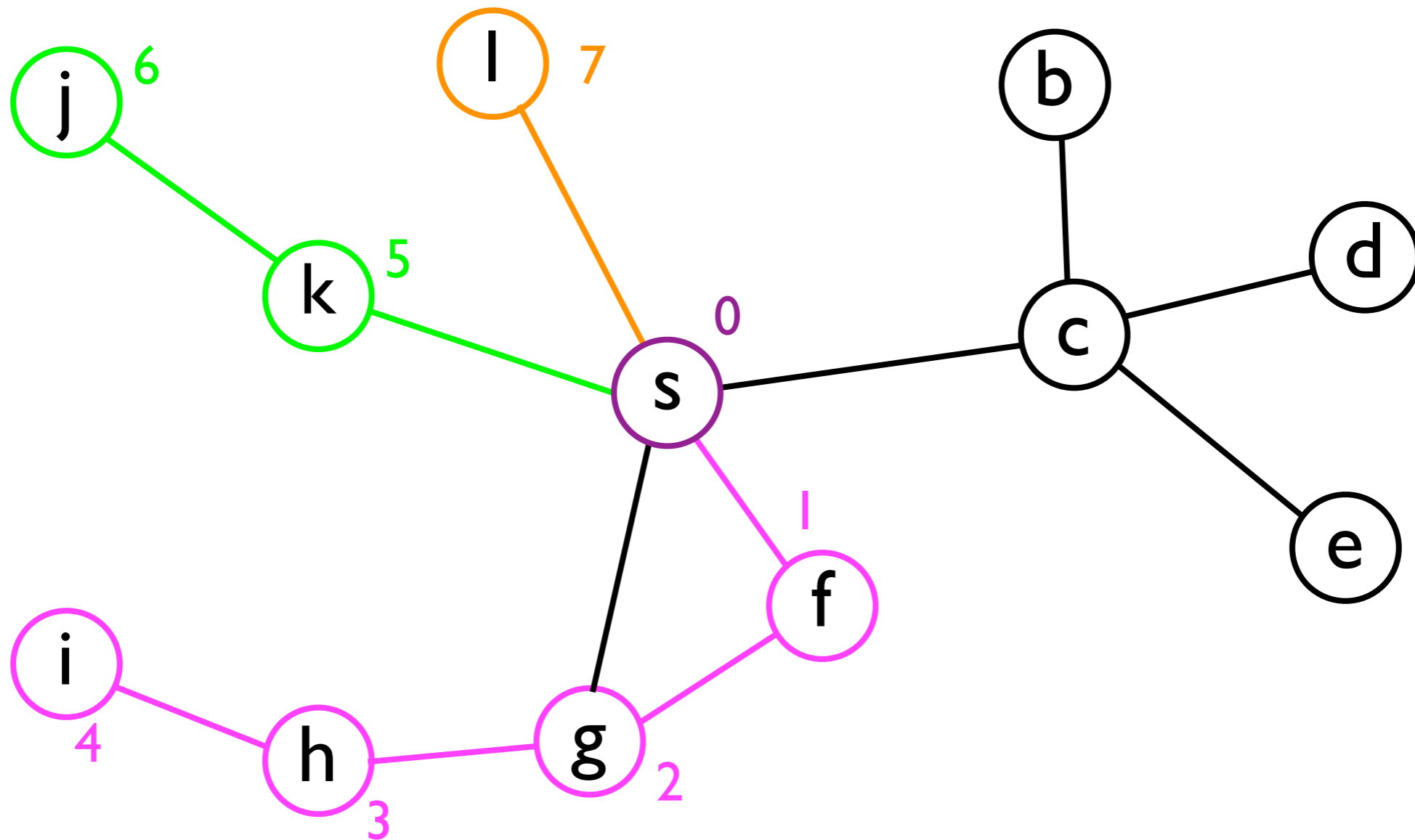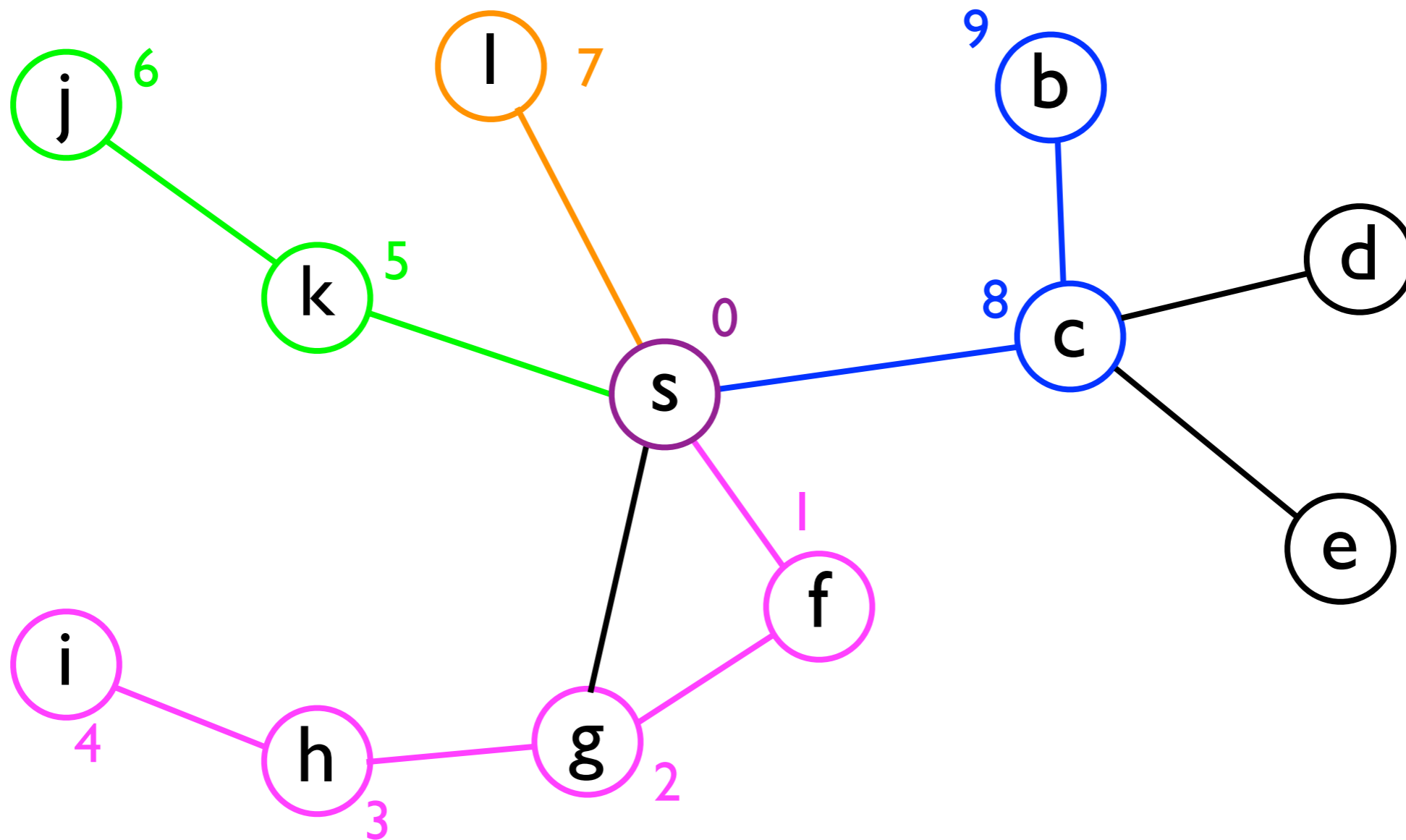
DFS

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

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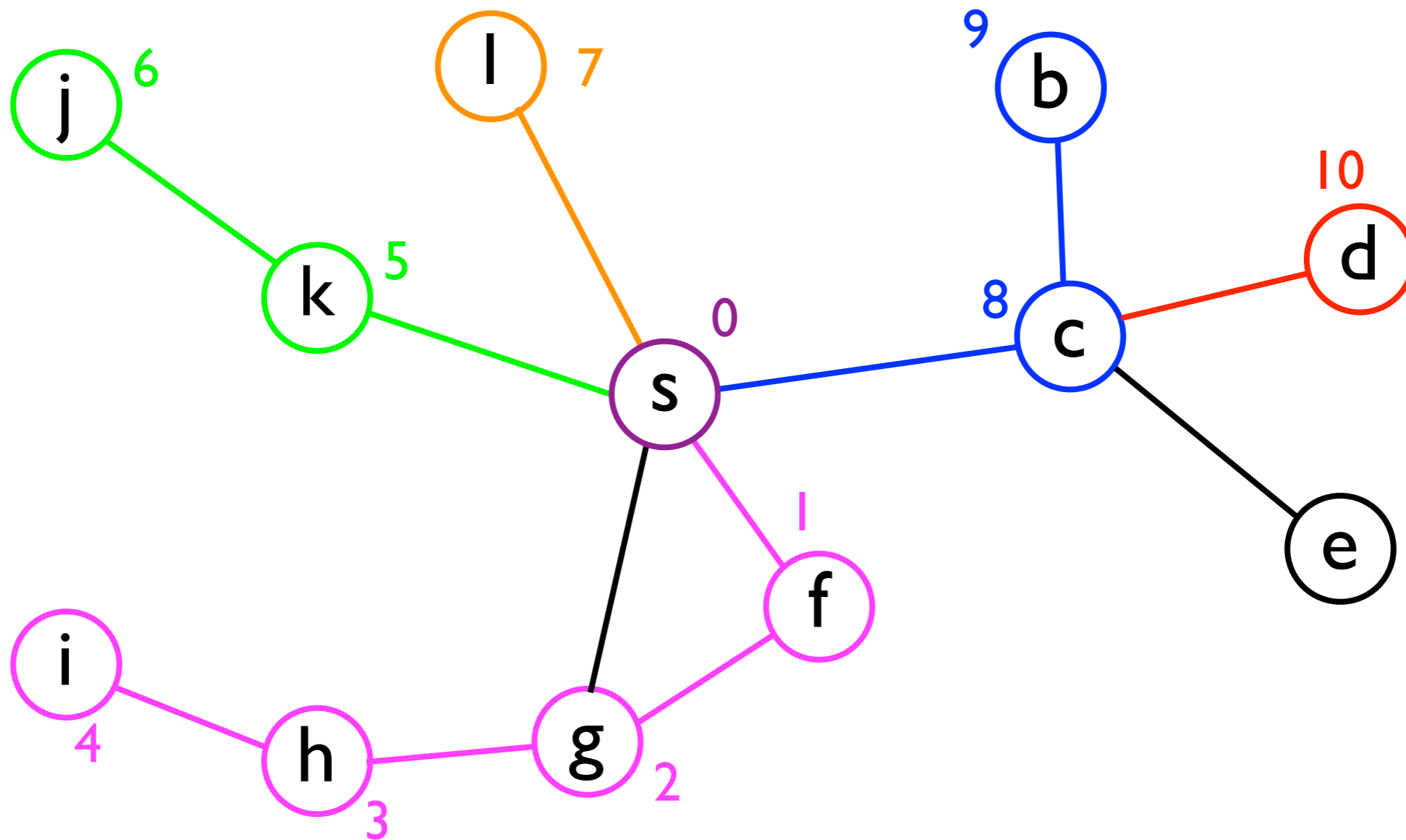
At the end of a path, it then "backtracks" and pursues another path "as far as it can go".
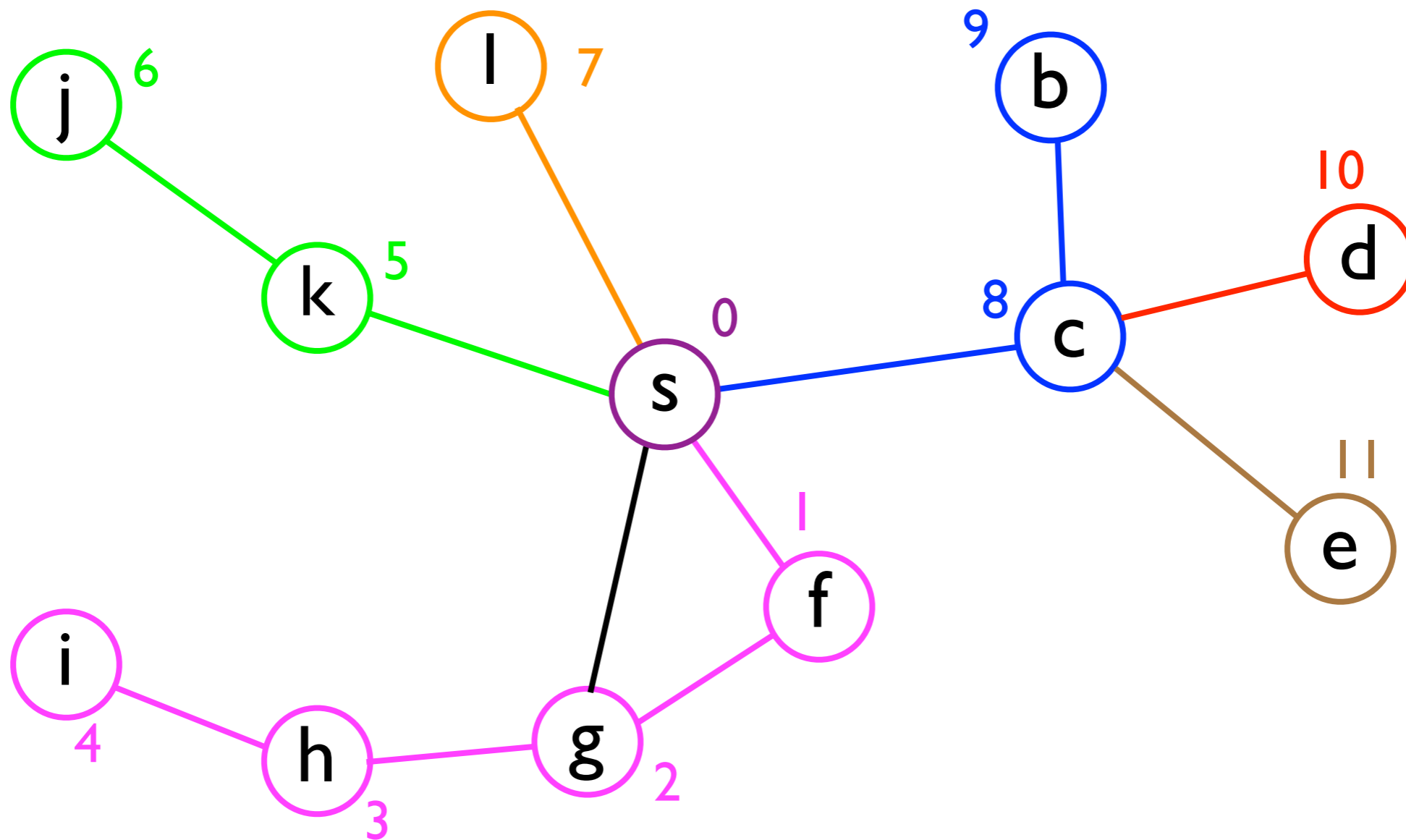
Implementing a graph

- Before describing the BFS and DFS algorithms in detail, it will be useful to define some "infrastructure" for dealing with graphs.

- Let us suppose we implement a `Graph` class using a *list of adjacency lists* as the underlying storage.

```
class Graph {
  int _numNodes;
  int[][] _adjacencyLists;
  ...
}
```

Implementing a graph

- We will assume that the nodes are numbered 0, 1, 2, ..., `_numNodes` - 1.

- To access the adjacency list of node *i*, we can fetch the *i*th adjacency list:

  ```
  final int[] adjacencyListForI = _adjacencyLists[i];
  ```

- The number of nodes to which node *i* is connected is then given by:

  ```
  int numNeighbors = adjacencyListForI.length;
  ```

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.

  - *Optional*: A *boolean array* to indicate which nodes have been/have not been visited.

    - Redundant, but faster than searching the list.

BFS implementation

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

```
List bfs (int s) {
   List visitedNodes;
   Queue nodesToVisit;

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

     for each neighbor n' of n:
       if nodesToVisit doesn't contain n':
         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`:   `nodesToVisit`:

s

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`:

c f g k l

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`:

f g k l

BFS in practice

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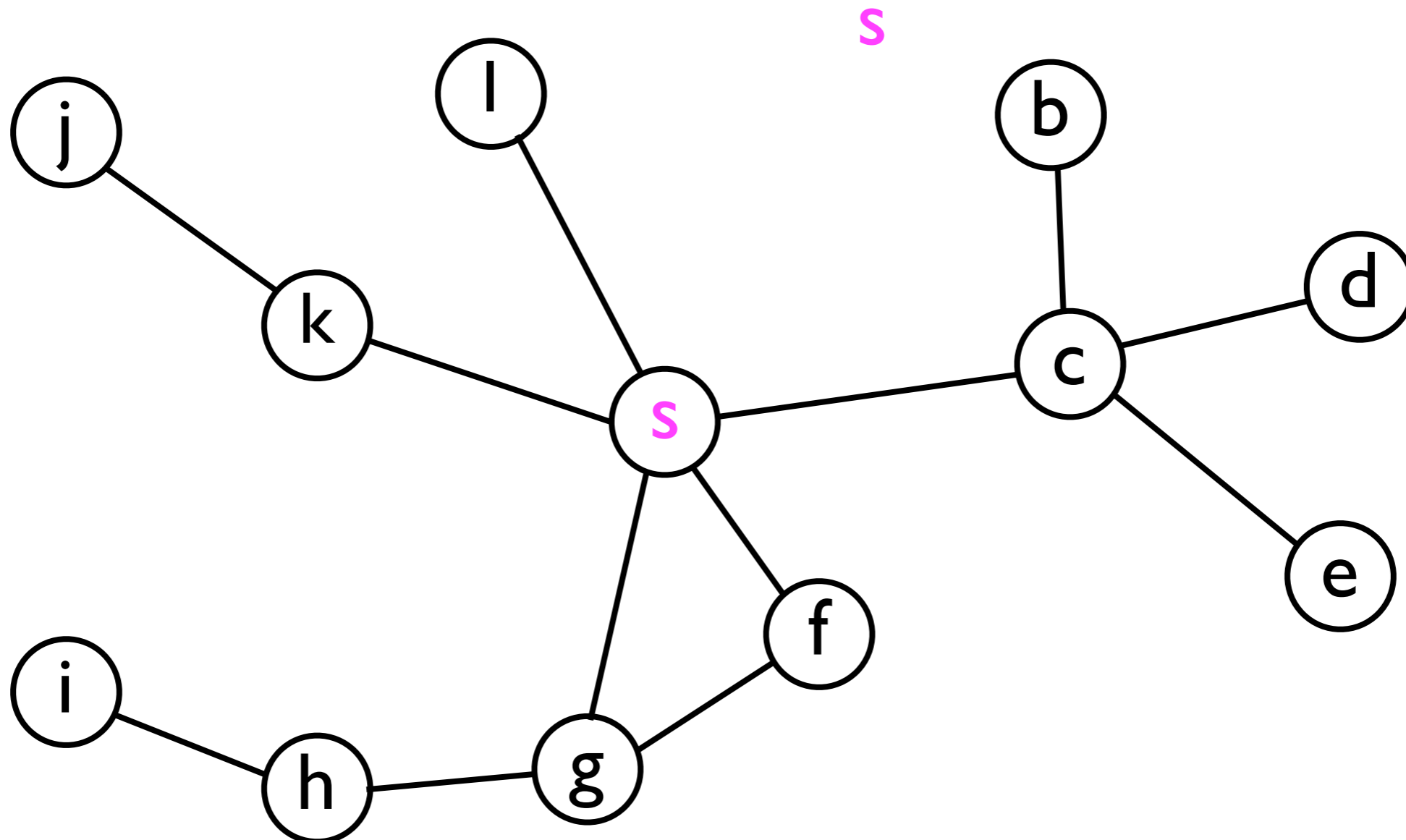
`visitedNodes`:

s c

`nodesToVisit`:

f g k l b d e

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
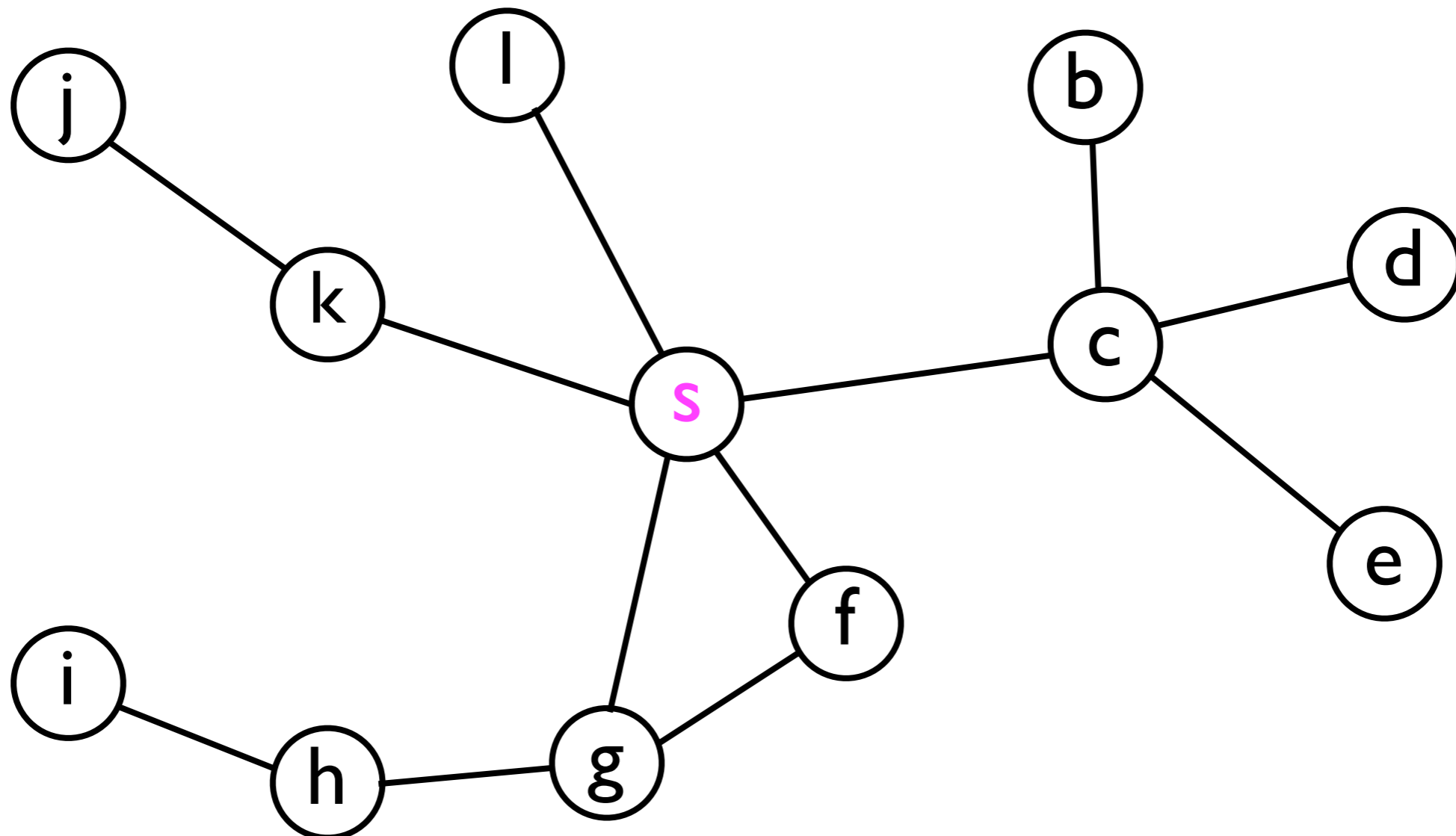
nodesToVisit:

g k l b d e

BFS in practice

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

`visitedNodes`:
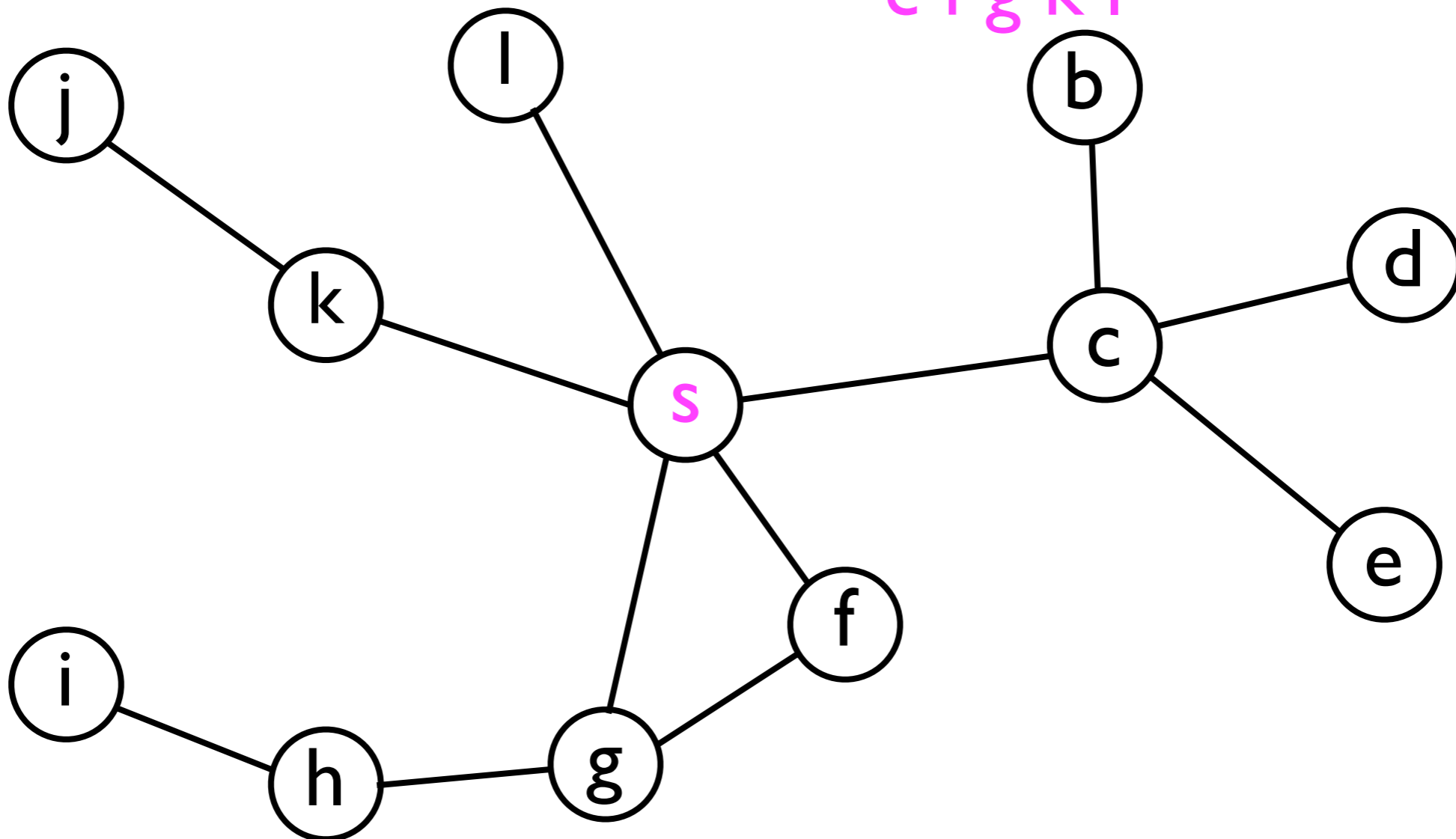
s c f g

`nodesToVisit`:

k l b d e

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`:

k l b d e h

BFS in practice

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

`visitedNodes`:
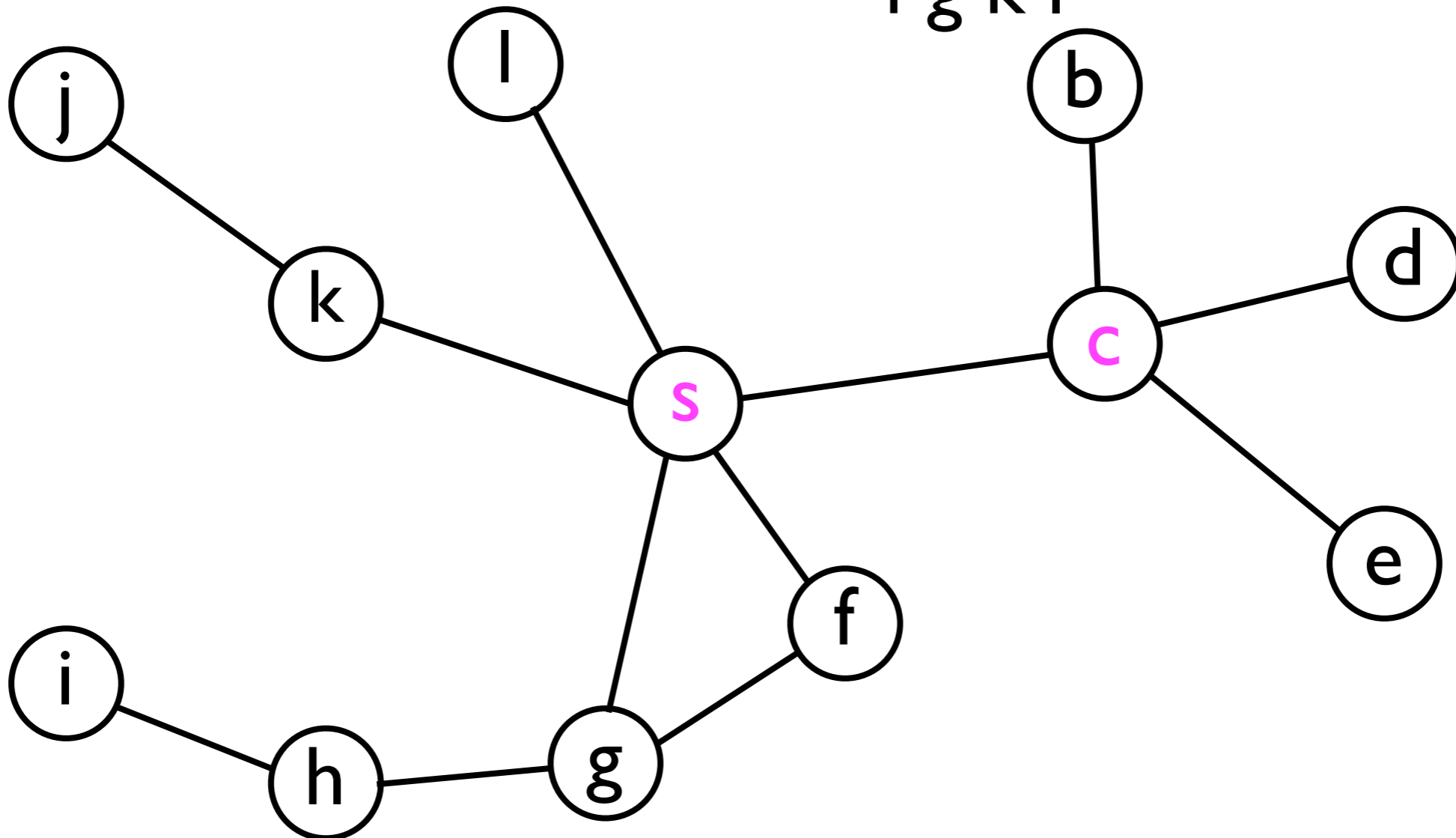
s c f g k

`nodesToVisit`:

l b d e h

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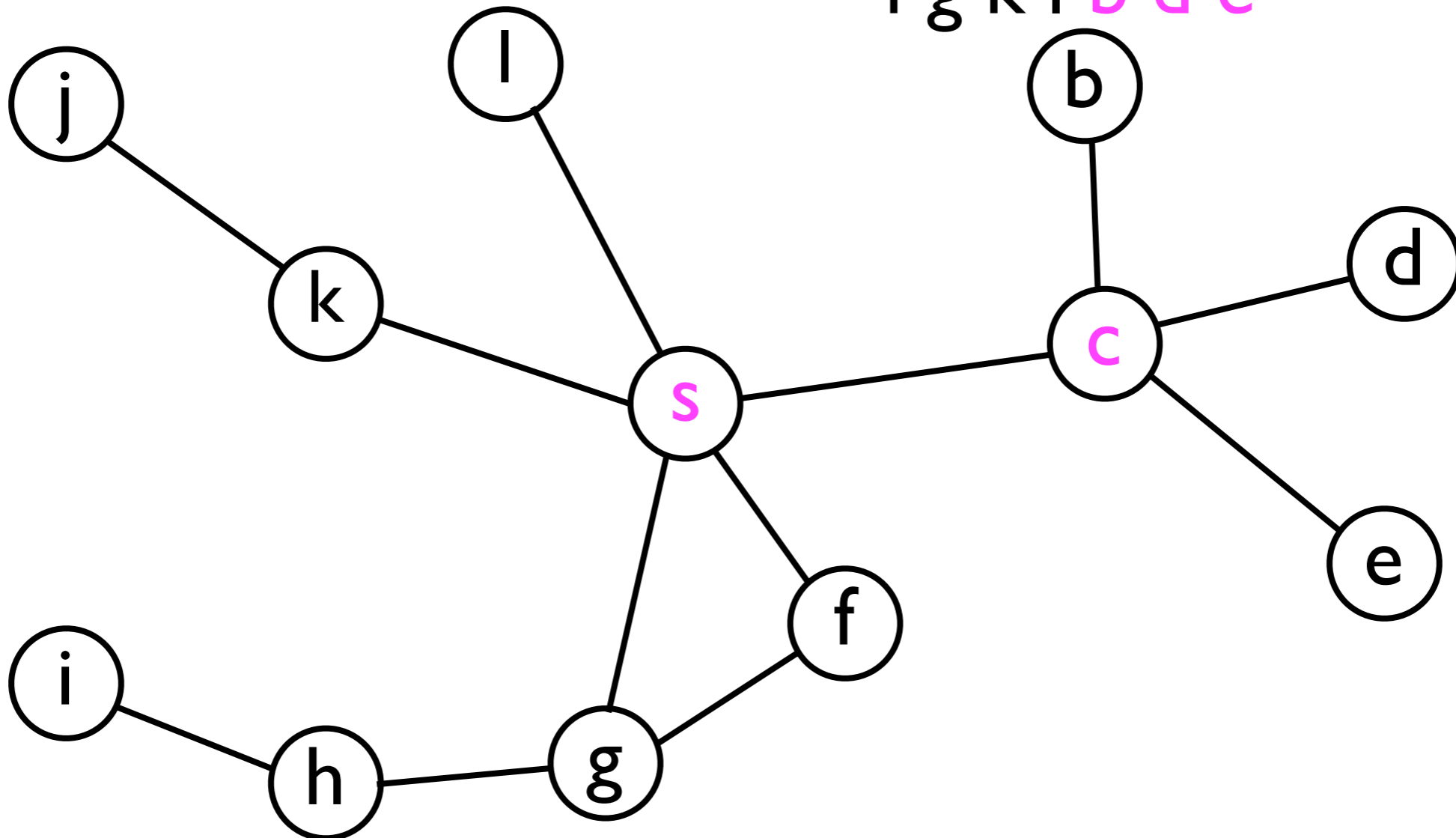
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

visitedNodes:

s c f g k

nodesToVisit:

l b d e h j

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`:

b d e h j

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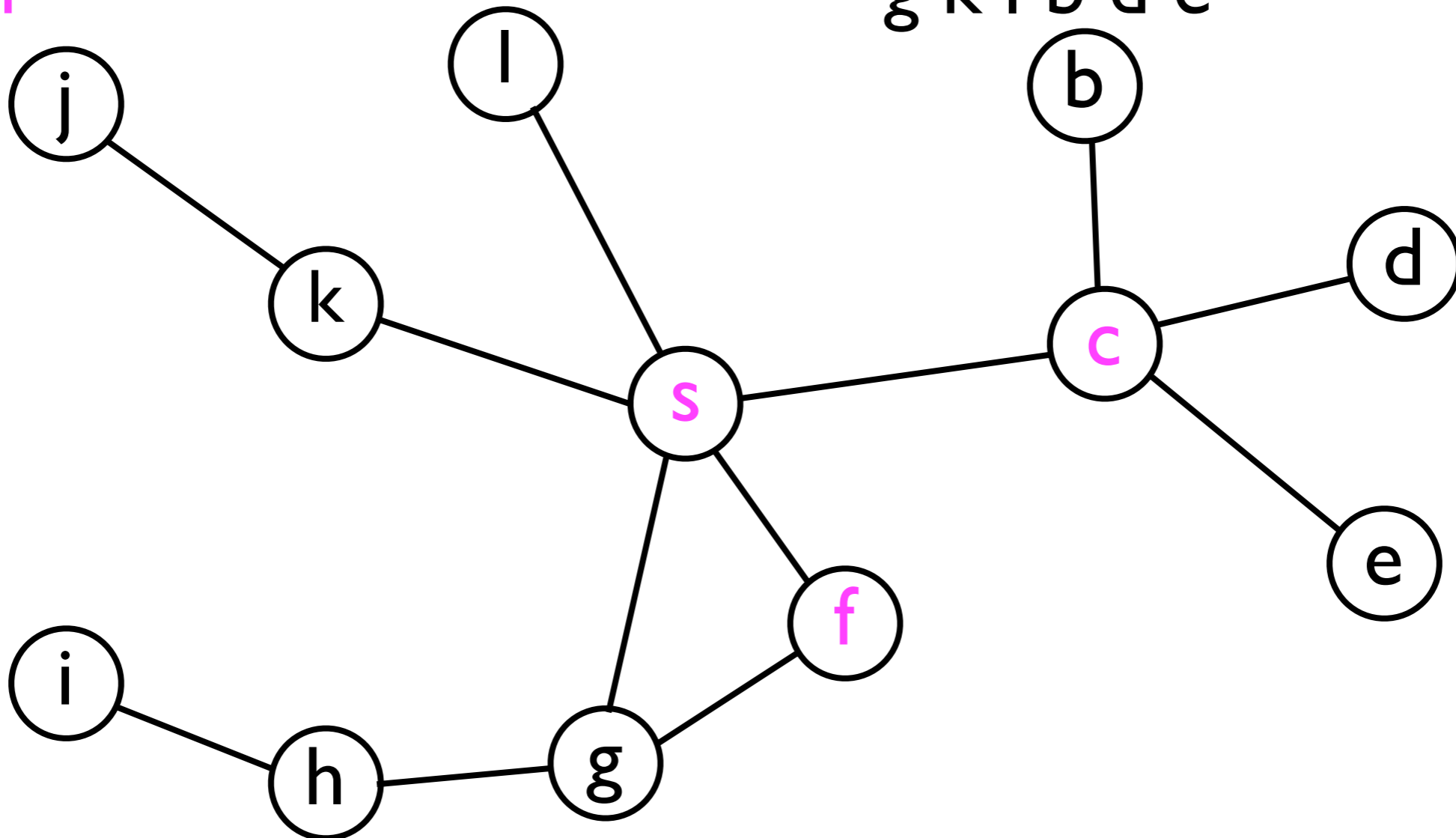
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes:`

s c f g k l b

`nodesToVisit:`

d e h j
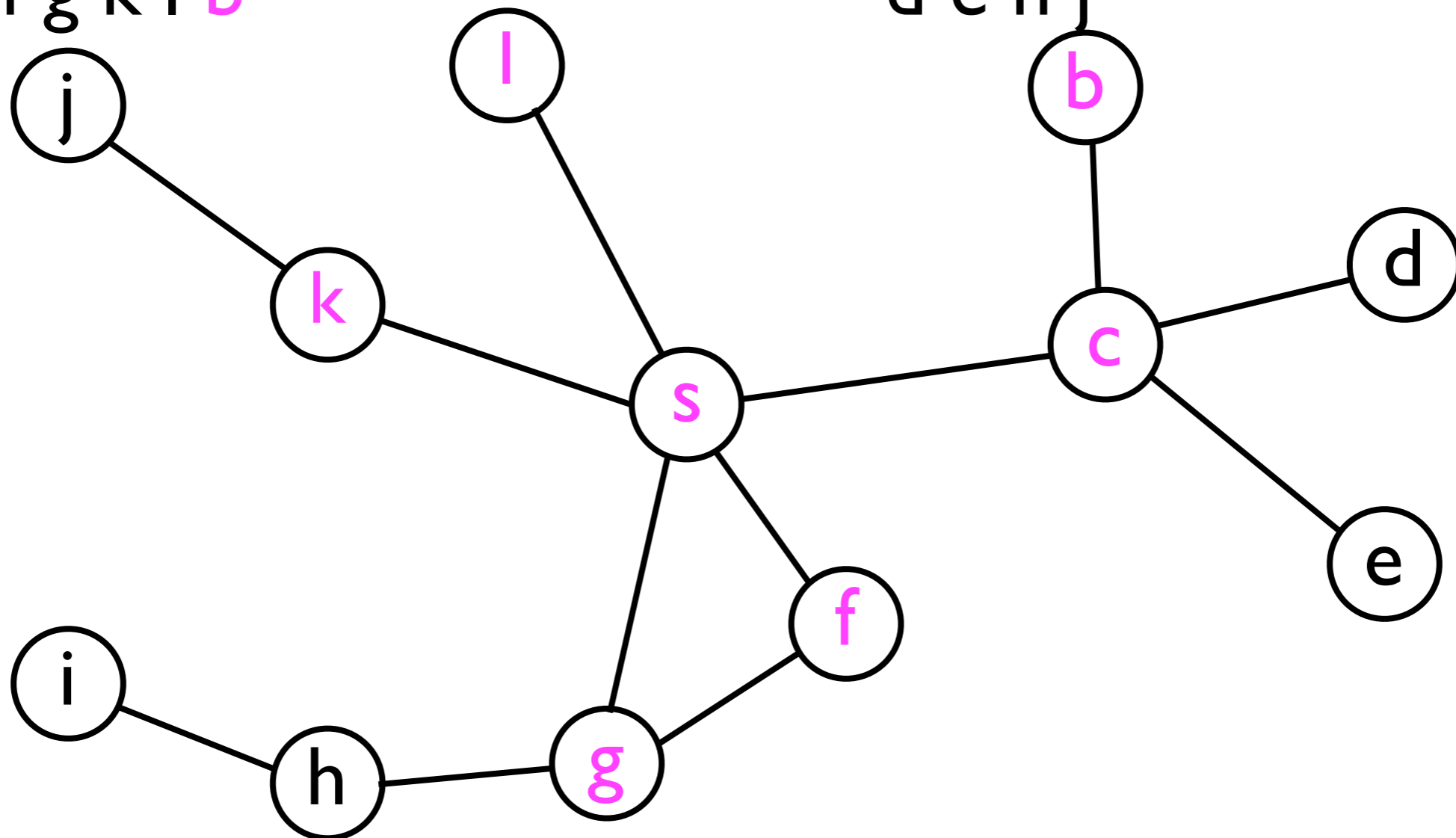
BFS in practice

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

visitedNodes:

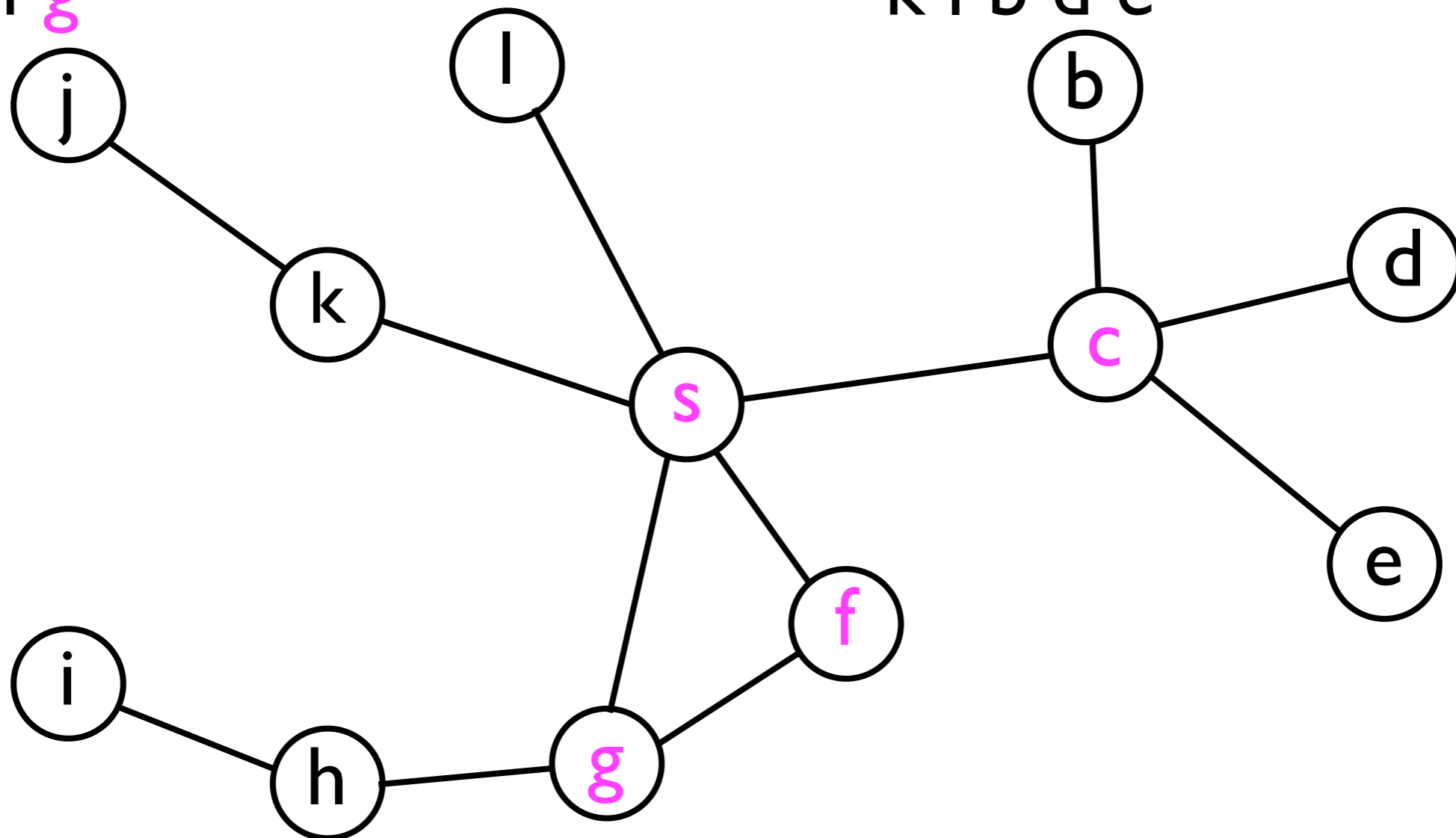s c f g k l b d

nodesToVisit:

e h j
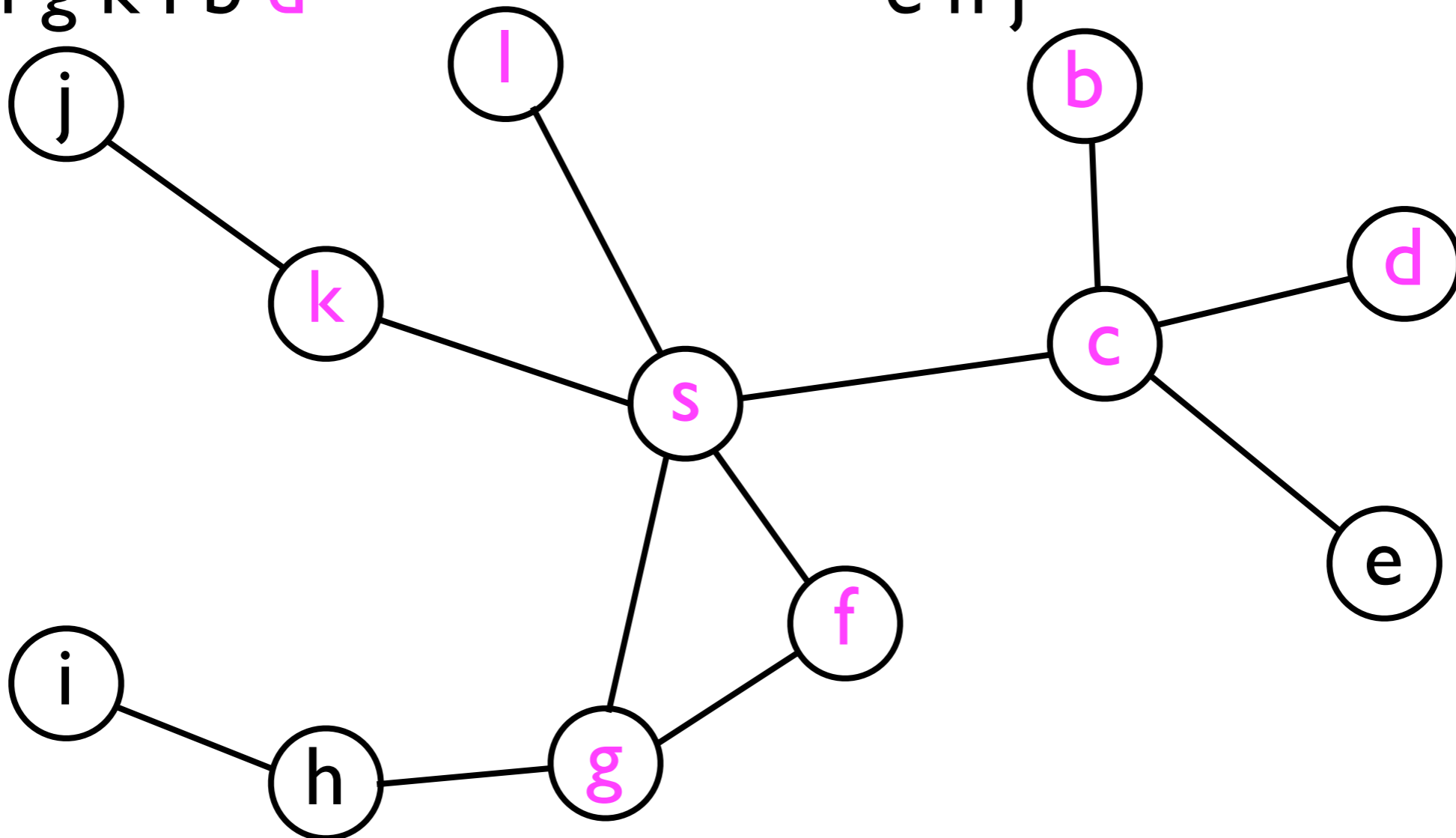
BFS in practice

- 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`:

h j
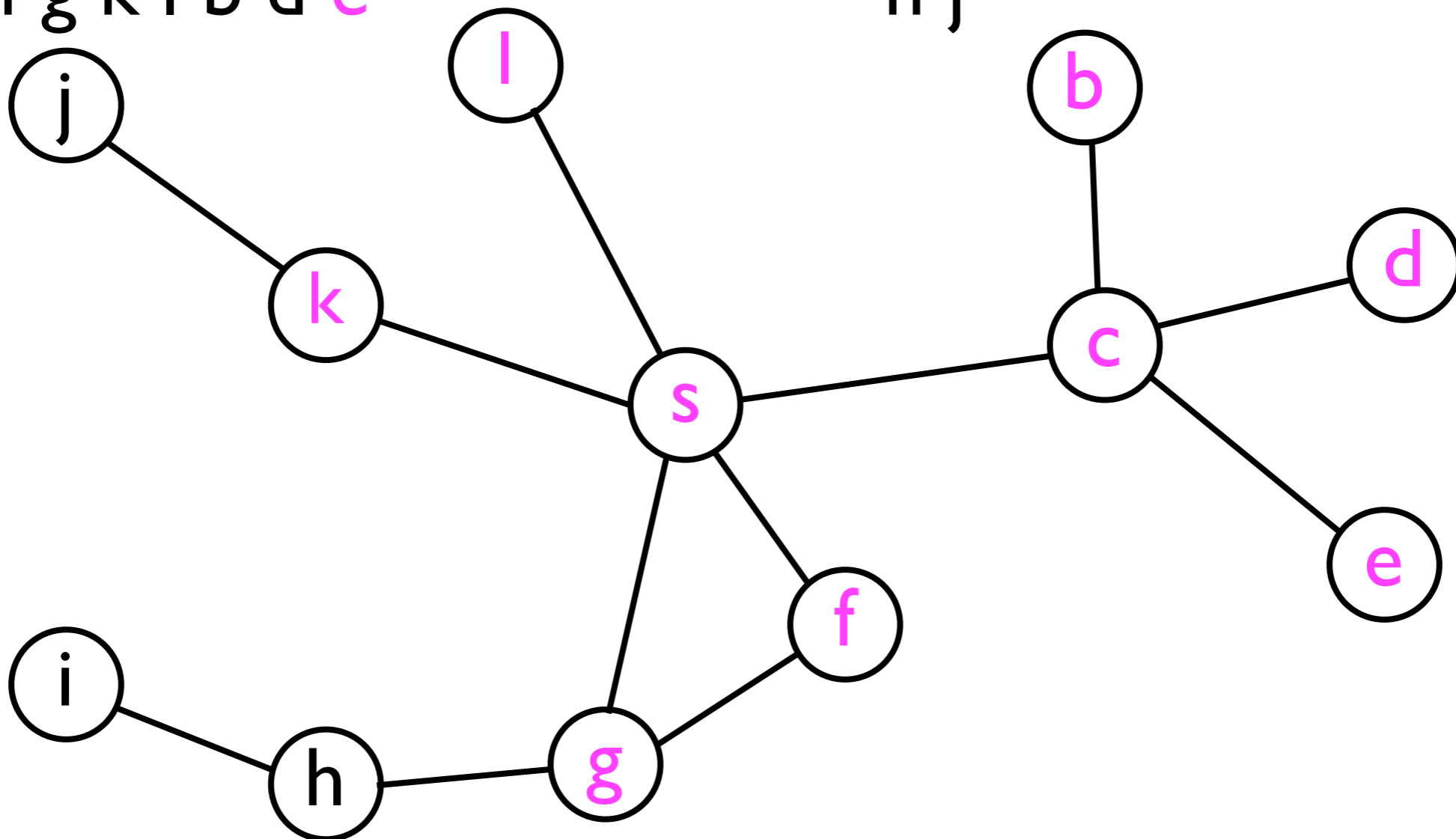
BFS in practice

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

`visitedNodes`:

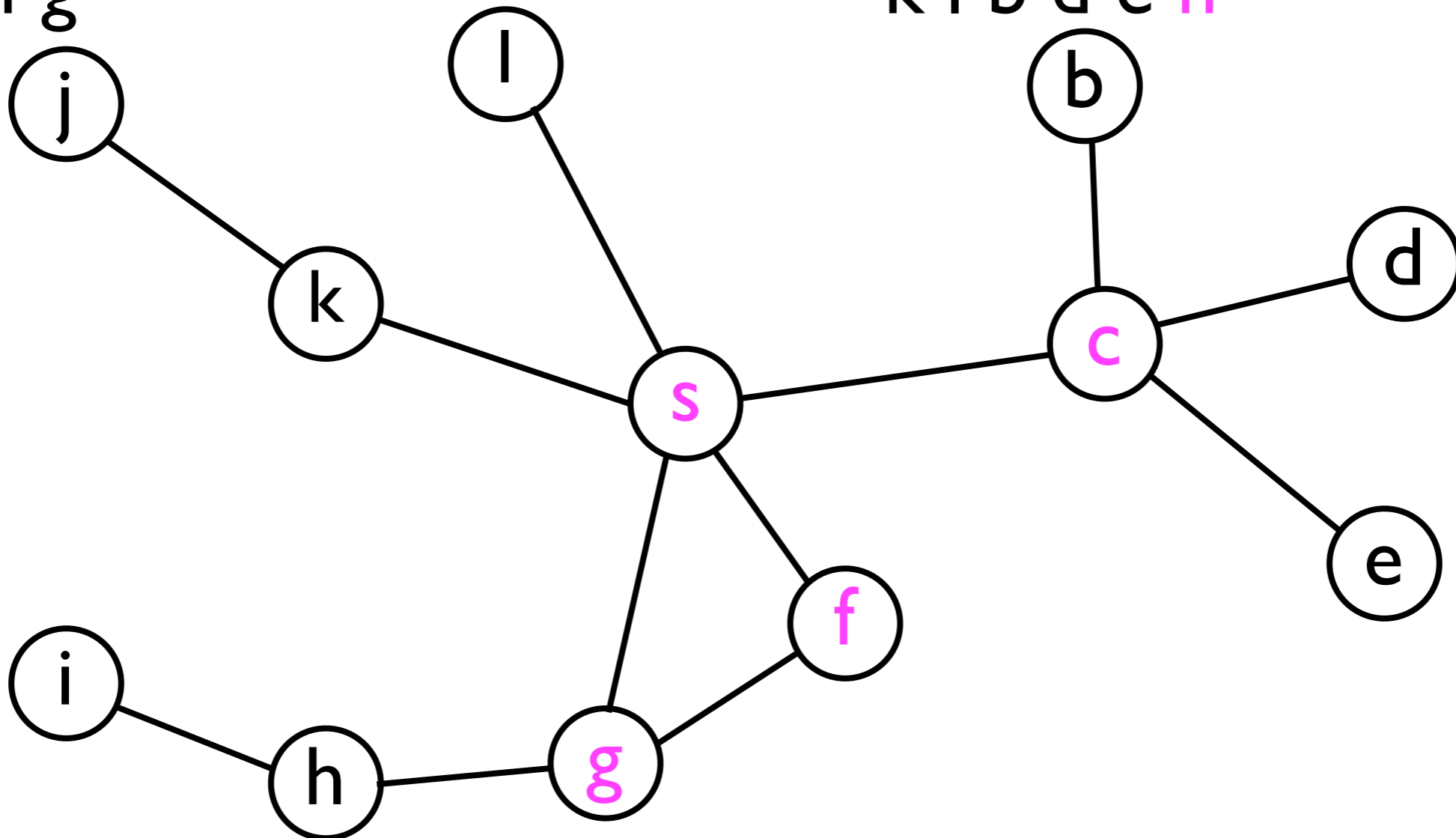s c f g k l b d e h

`nodesToVisit`:

j
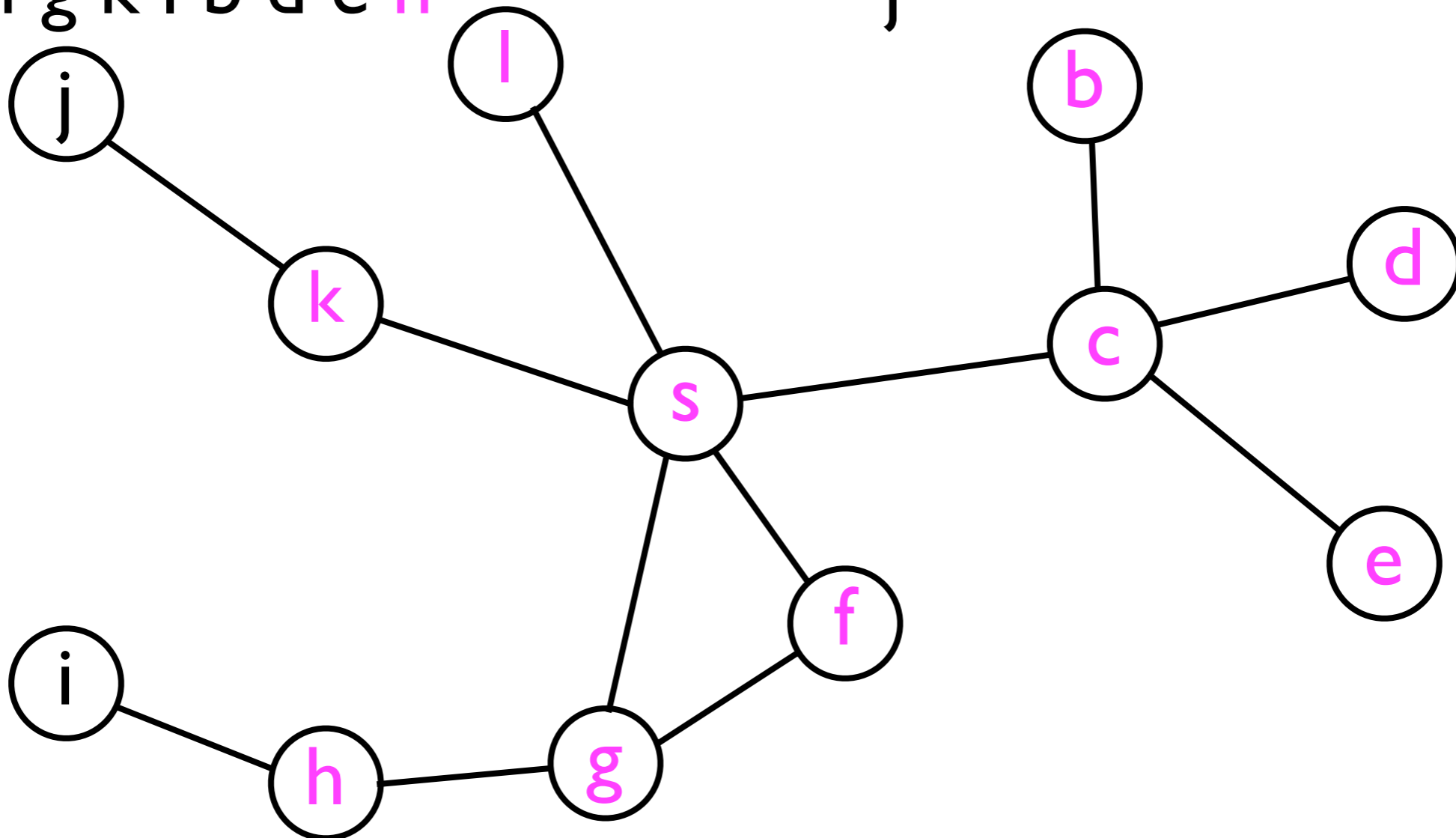
BFS in practice

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

`visitedNodes`:

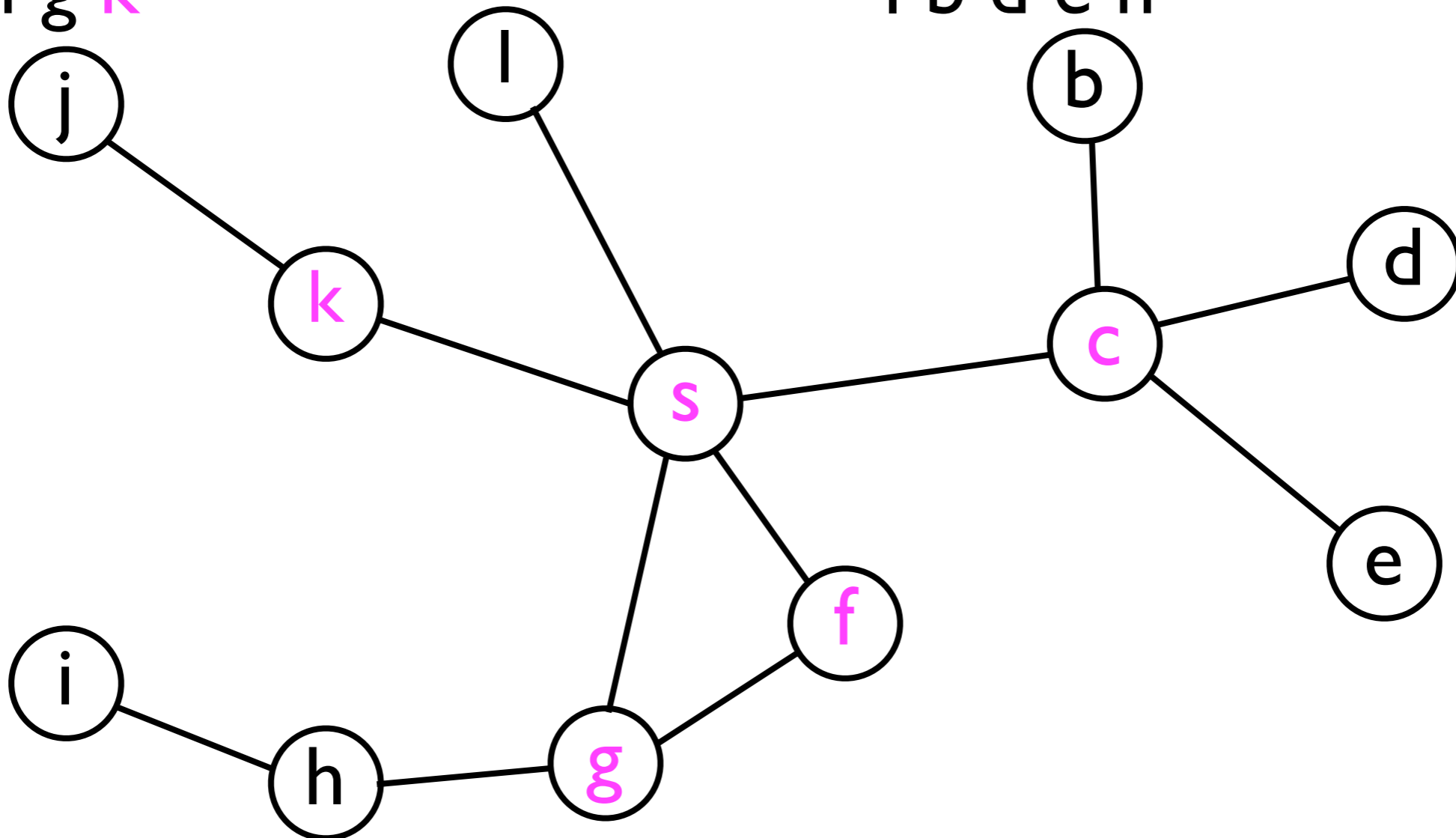s c f g k l b d e h

`nodesToVisit`:

j i
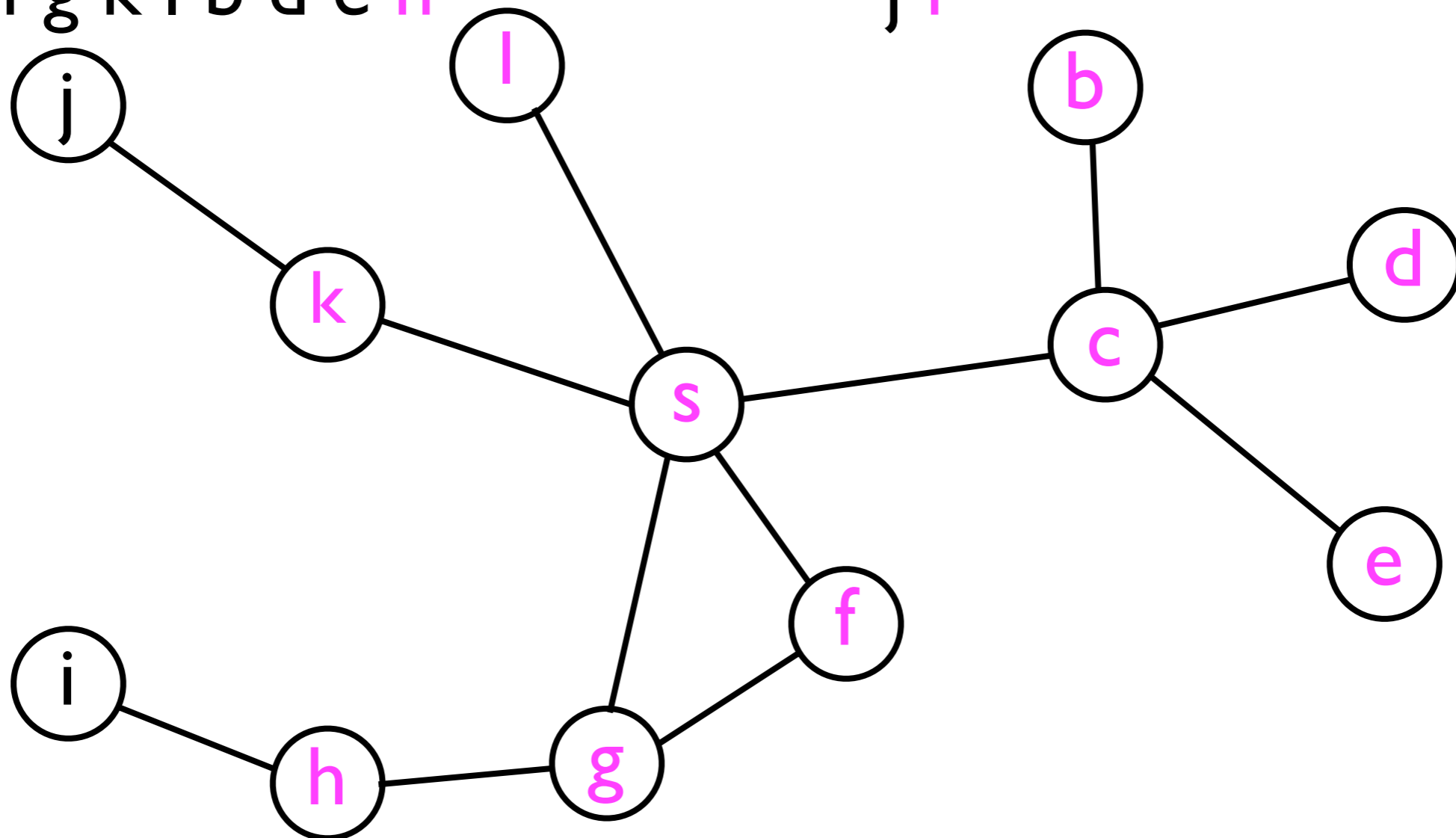
BFS in practice

- 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`:

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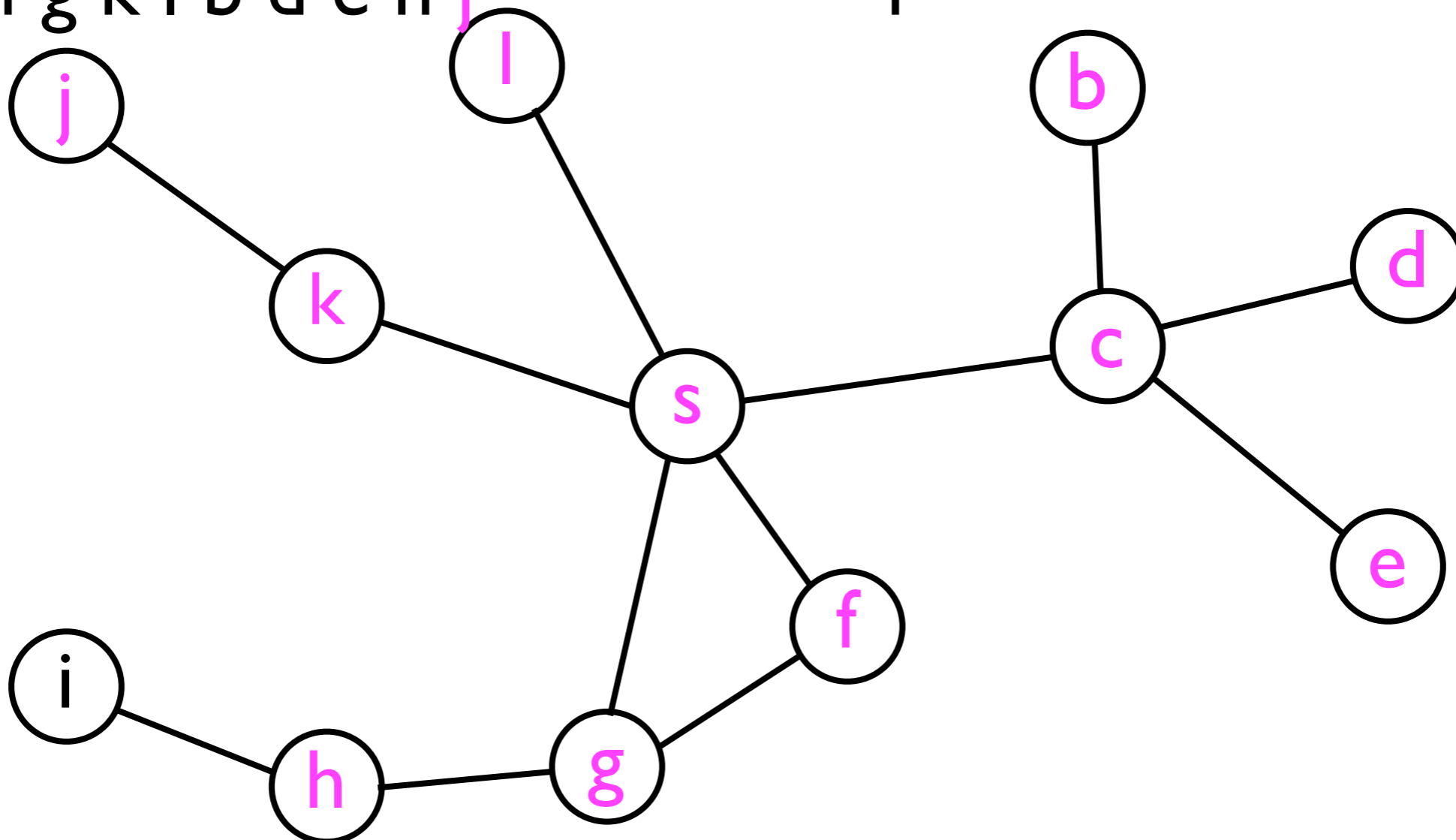
BFS in practice
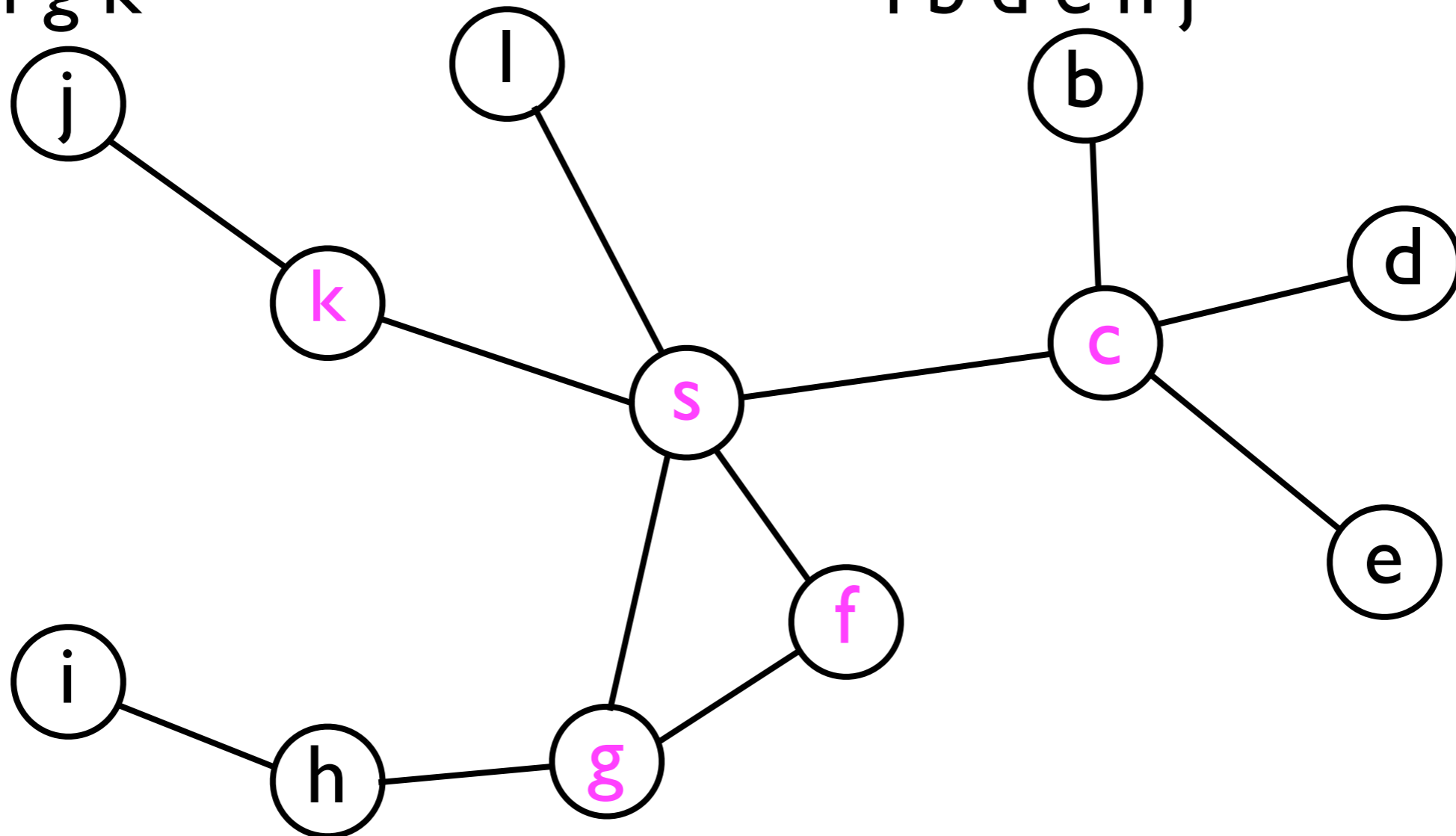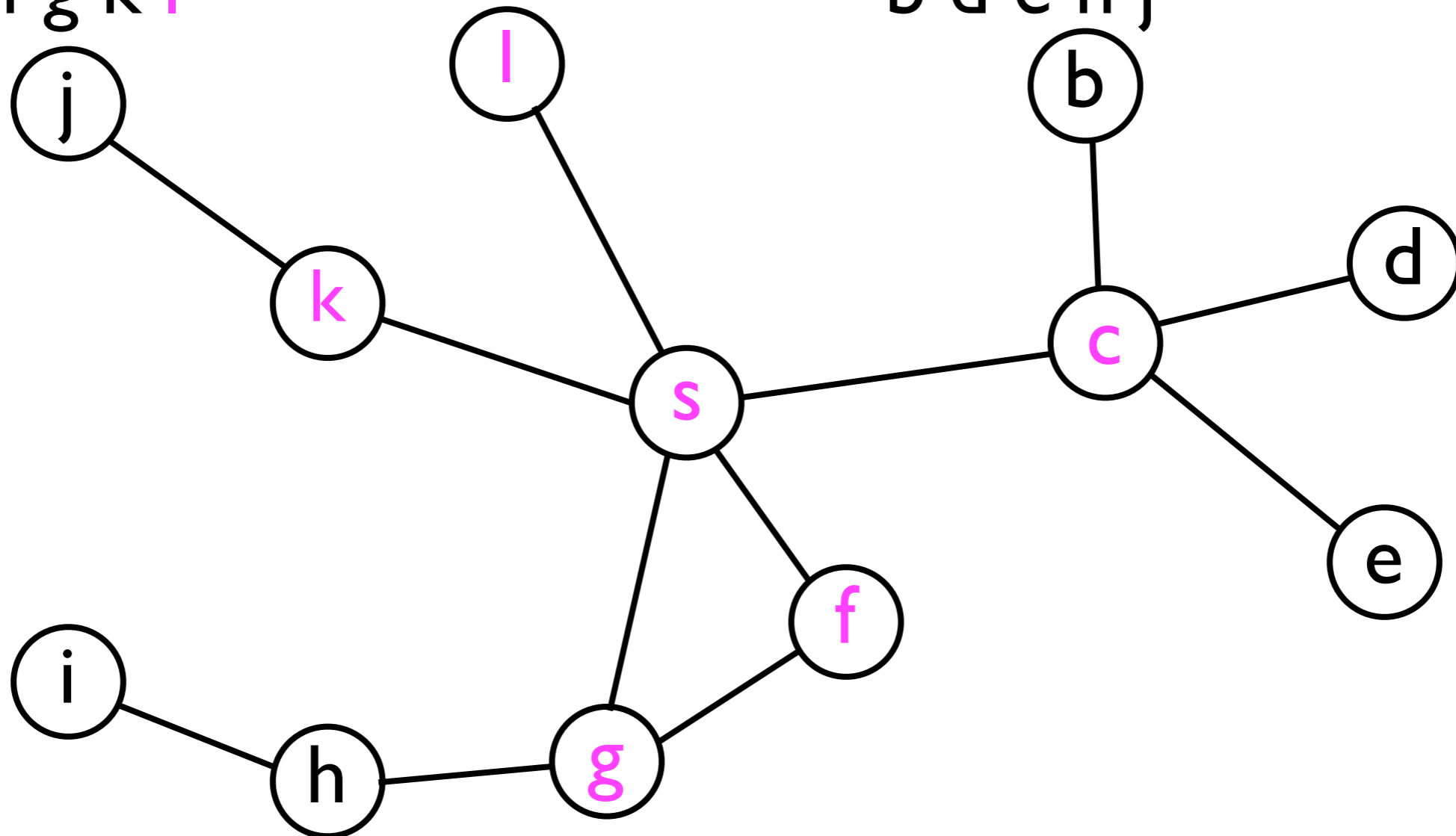
- Let's look at how the `visitedNodes` and `nodesToVisit` data structures are updated when exploring the graph from earlier...

`visitedNodes`:   `nodesToVisit`:

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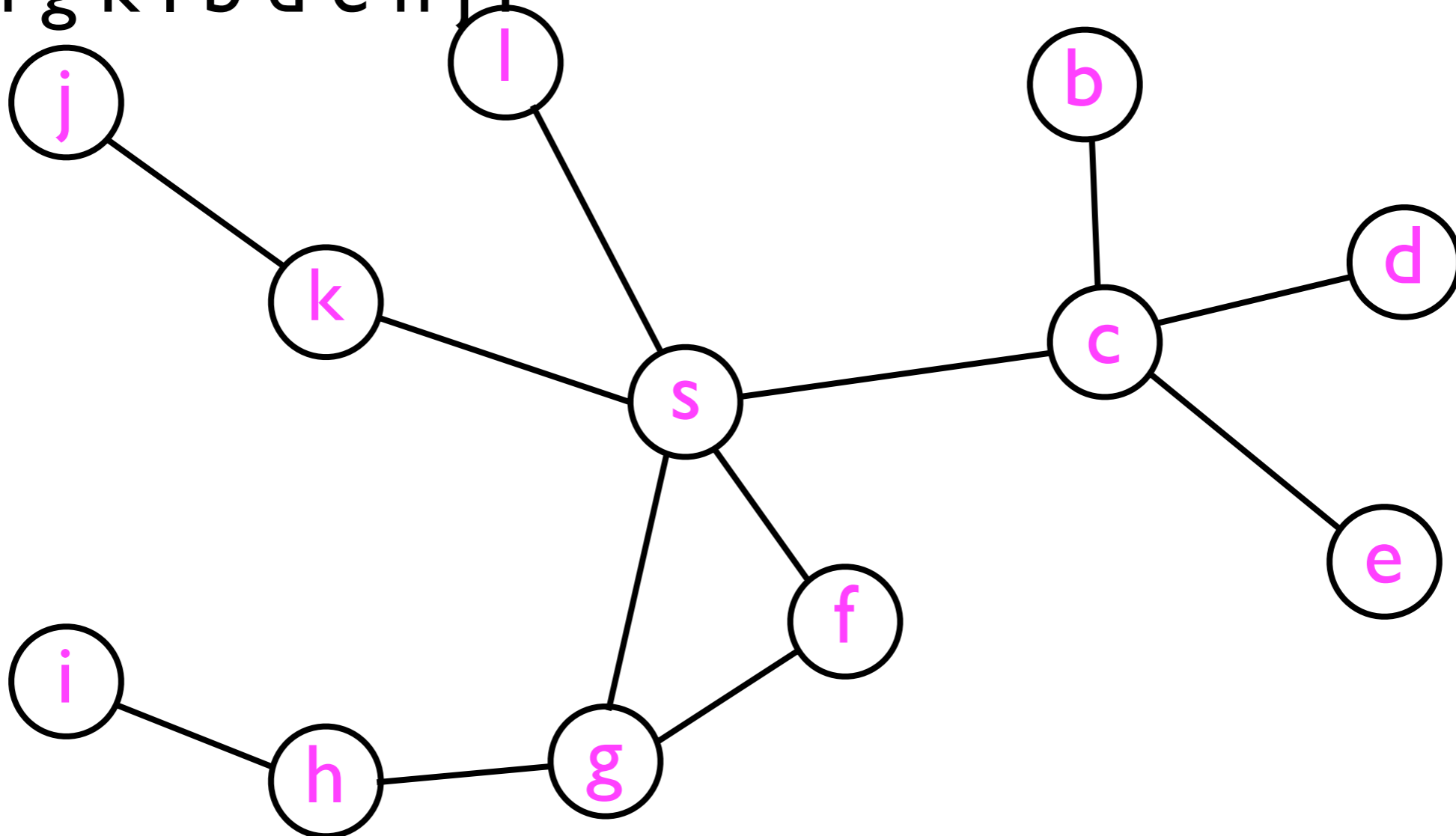
DFS implementation

- 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 dfs (int s) {
   List visitedNodes;
   Stack nodesToVisit;

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

     for each neighbor n' of n:
        if nodesToVisit doesn't contain n':
          nodesToVisit.push(n');
   }

   return visitedNodes;
}
```

Finding shortest paths

- It is (relatively) straightforward to modify the basic BFS method to give us a *shortest path* between any two nodes *s* and *t*

- Here we only "sketch" a solution:

  - Whenever we visit a node `n'` during BFS, we record its distance from `s` in an array.

    ```
    distances[n'] = distances[n] + 1;
    ```

  - As soon as the BFS discovers node *t*, the BFS terminates.

Finding shortest paths

- When the modified BFS terminates, if `distances[t] == k`, then there exists a *k*-step path from *s* to *t*.

- Since BFS visits *all* nodes that are reachable from s in *k* steps before examining nodes that are reachable from s in *k*+1 steps, we know that *k* is the minimal path length from *s* to *t*.

  - I.e., we've found a *shortest path*.

  - Note that there can be *multiple* shortest paths (all of length *k*) from *s* to *t*.

Finding shortest paths

- To determine the nodes along this shortest path, we have to *backtrack*:

    - Start at node *t*.

    - Examine each of *t*'s neighbors:

        - Find a neighbor *n* whose distance from *s* is only *k*-1.

            - Add *n* to the "shortest path" from *s* to *t*.

    - Keep backtracking from *n* back to *s*.

Finding shortest paths

- For instance, suppose the modified BFS computed the following distances for the nodes reachable from $s=0$:

```
distances[0] = 0
distances[1] = 1
distances[2] = 2
distances[3] = 3
```

- Then, to find the shortest path from $s=0$ to $t=2$, we *backtrack* starting at t.

Finding shortest paths

- For instance, suppose the modified BFS computed the following distances for the nodes reachable from $s=0$:

```
distances[0] = 0
distances[1] = 1
distances[2] = 2
distances[3] = 3
```

- Then, to find the shortest path from $s=0$ to $t=2$, we *backtrack* starting at t.

If $t=2$ is 2 away from s, then which of t's neighbors is only 1 away from $s=0$?

Path: 2

Finding shortest paths

- For instance, suppose the modified BFS computed the following distances for the nodes reachable from $s=0$:

```
distances[0] = 0
distances[1] = 1
distances[2] = 2
distances[3] = 3
```

- Then, to find the shortest path from $s=0$ to $t=2$, we *backtrack* starting at t.

If $t=2$ is 2 away from s, then which of t's neighbors is only 1 away from $s=0$?

Path: 2 1

Finding shortest paths

- For instance, suppose the modified BFS computed the following distances for the nodes reachable from $s=0$:

```
distances[0] = 0
distances[1] = 1
distances[2] = 2
distances[3] = 3
```

- Then, to find the shortest path from $s=0$ to $t=2$, we *backtrack* starting at t.

n

If $n=1$ is 1 away from s, then which of t's neighbors is only 0 away from $s=0$?

Path: 2 1

Finding shortest paths

- For instance, suppose the modified BFS computed the following distances for the nodes reachable from $s=0$:

```
distances[0] = 0
distances[1] = 1
distances[2] = 2
distances[3] = 3
```
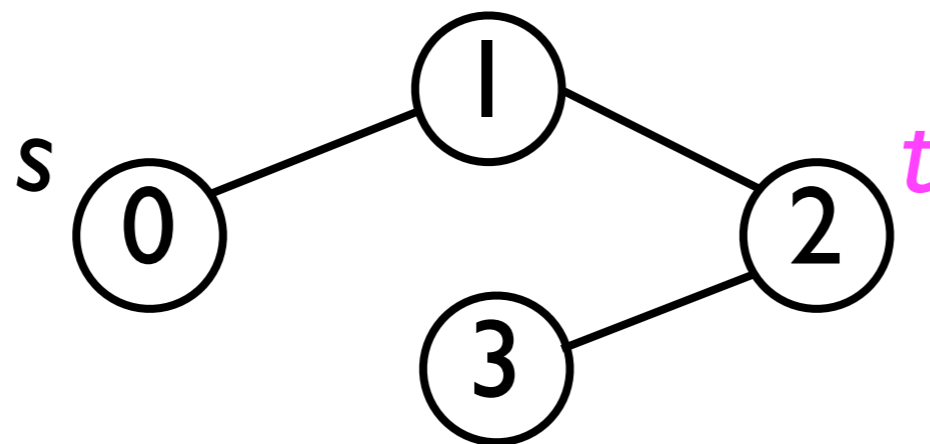
- Then, to find the shortest path from $s=0$ to $t=2$, we *backtrack* starting at t.

If $n=1$ is 1 away from s, then which of t's neighbors is only 0 away from $s=0$?
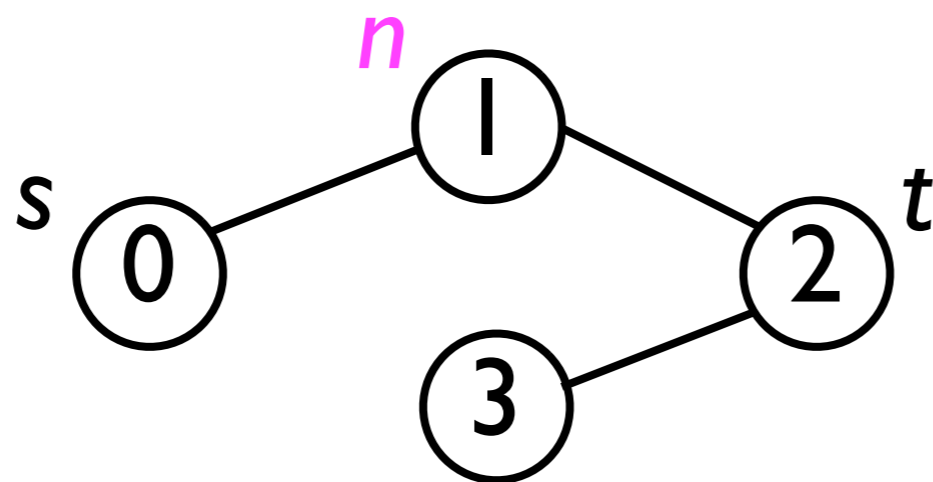
Path: 2 1 0

Finding shortest paths

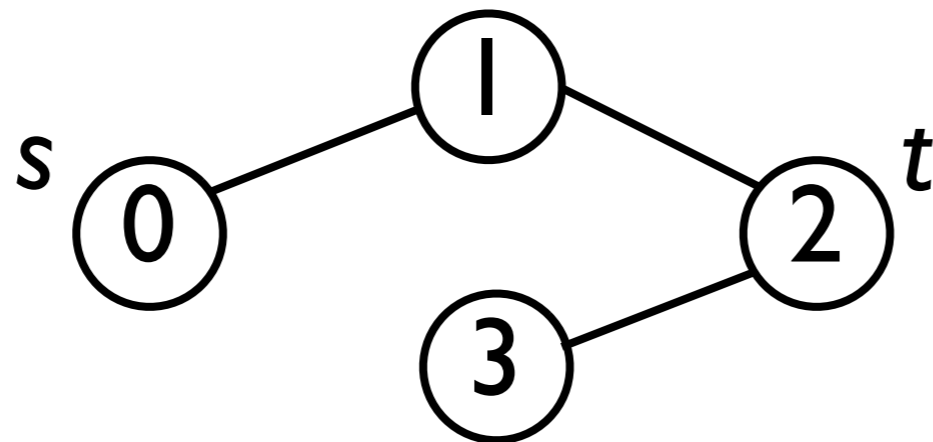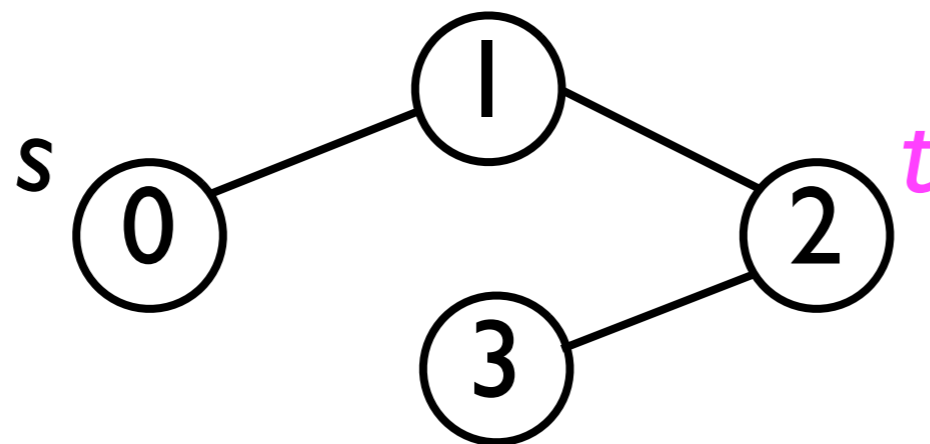- For instance, suppose the modified BFS computed the following distances for the nodes reachable from $s=0$:

```
distances[0] = 0
distances[1] = 1
distances[2] = 2
distances[3] = 3
```

- Then, to find the shortest path from $s=0$ to $t=2$, we *backtrack* starting at t.
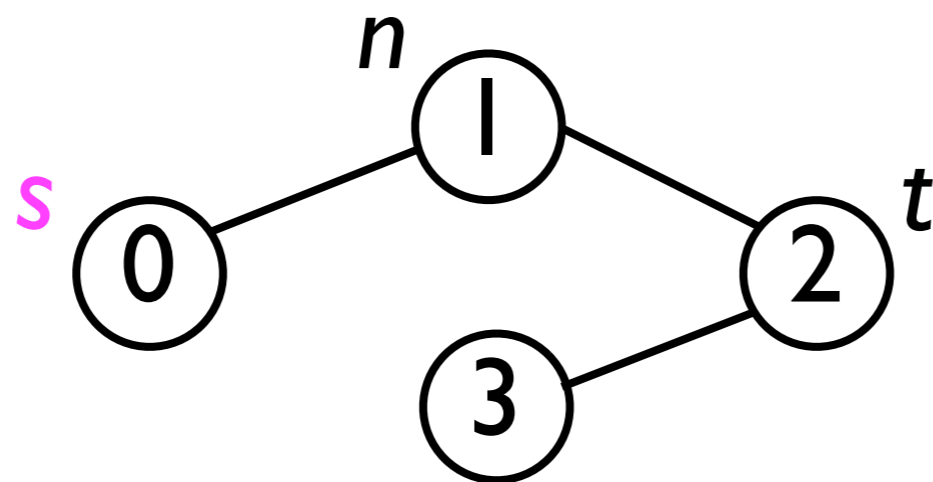
Path: 0 1 2

Kevin Bacon Number demo.

Sorting.

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 $x \leq y$ and $y \leq z$, then $x \leq z$.

    - **Anti-symmetric**: if $x \leq y$ and $y \leq x$, 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**: $e_1 \leq e_2 \leq e_3 \leq ... \leq e_{n-1} \leq e_n$

  - **Non-increasing**: $e_1 \geq e_2 \geq e_3 \geq ... \geq e_{n-1} \geq e_n$

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*, 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 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 amount of operations, it is always possible that Bogosort hasn't "found" the right permutation yet.

Bogosort

- The 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 may not yet be sorted.

Approach 1: sorted- and unsorted- parts

- For example, consider the following array:

$$6 \quad 1 \quad 3 \quad 2 \quad \textcolor{magenta}{4 \quad 5 \quad 7 \quad 8}$$

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:

  Unsorted part   Sorted part

  6 1 4 3 8 7 2 5

  6 1 4 3 7 2 5   Find and move largest   8

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   Find and move largest ⟶   7 8

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   Find and move largest   6 7 8

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 of unsorted part
    Add it at left-most position of sorted part
    Remove it from unsorted part
}
```

- 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.

- Adding and removing takes $O(1)$ time.

- Total time cost: $1 + 2 + ... + n + n*O(1) = n(n-1)/2 + O(n)$
$$= O(n^2)$$

Selection sort

- Pseudocode:

```
void selectionSort (int[] array) {
  While size of unsorted part > 0:
    Find largest element of unsorted part
    Add it at left-most position of sorted part
    Remove it from unsorted part
}
```

- 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.