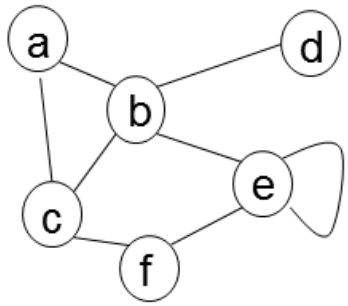
Graph Traversals

CS1113

Paths

A path in a graph is a sequence of oriented edges, such that the endpoint of one oriented edge is the startpoint of the next oriented edge in the sequence.

Examples:



<(*a*,*b*), (*b*,*e*),(*e*,*f*),(*f*,*c*)> is a path
<(*a*,*b*),(*b*,*c*),(*c*,*a*)> is a path
<(*f*,*e*),(*e*,*e*),(*e*,*f*),(*f*,*c*)> is a path

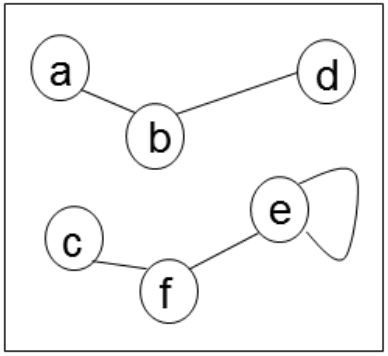
<(*d*,*b*),(*f*,*c*)> is not a path
<(*c*,*f*),(*e*,*f*)> is not a path
<(*d*,*b*),(*c*,*f*),(*b*,*c*)> is not a path
<(*c*,*b*),(*b*,*a*),(*b*,*e*)> is not a path
<(*c*,*f*),(*f*,*d*)> is not a path

What about (i) <(b,c),(c,f),(f,e),(e,e),(e,b)>?
(ii) <(f,c),(c,b),(a,b),(b,d)>;?
(iii) <(c,a)>?

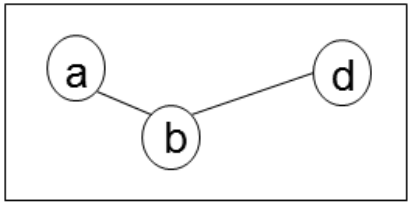
CS1113

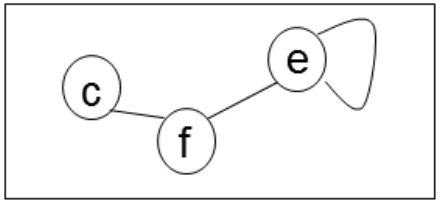
Connected graphs

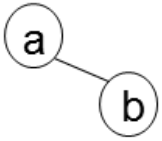
A connected component, H, of a graph G is a connected subgraph of G that is not a proper subgraph of any other connected subgraph of G.
(we could say H is a maximal connected subgraph of G)



has two connected components:

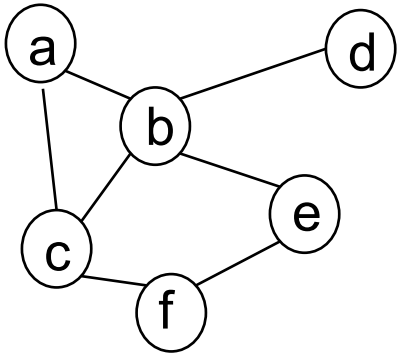






is not a connected component (it is a connected subgraph, but not a maximal connected subgraph)

Graphs and connectivity



- can we get from A to B by following edges?
- where can we reach by following edges from X?
- is there a path from X to every other vertex?
- what is the shortest path from X to Y?
- what are all the connected components of the graph?
- is there a single vertex whose removal would disconnect the graph?
- what is the smallest set of vertices that we need to remove to disconnect A and B?
- can we find two paths from A and B that share no edges or intermediate vertices?
- which vertex is 'central' to the graph?

Graph Traversals

Most of these questions can be answered by algorithms which *traverse* the graph

- visit all the vertices that can be reached from some starting vertex, by moving across edges

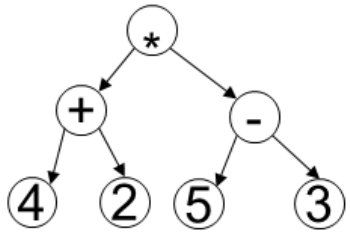
We will start with the two basic traversal algorithms:

- depth-first search
- breadth-first search

Preorder traversal

to visit a node:
read the element then visit the children

'preorder' because we do the parent's element before the children

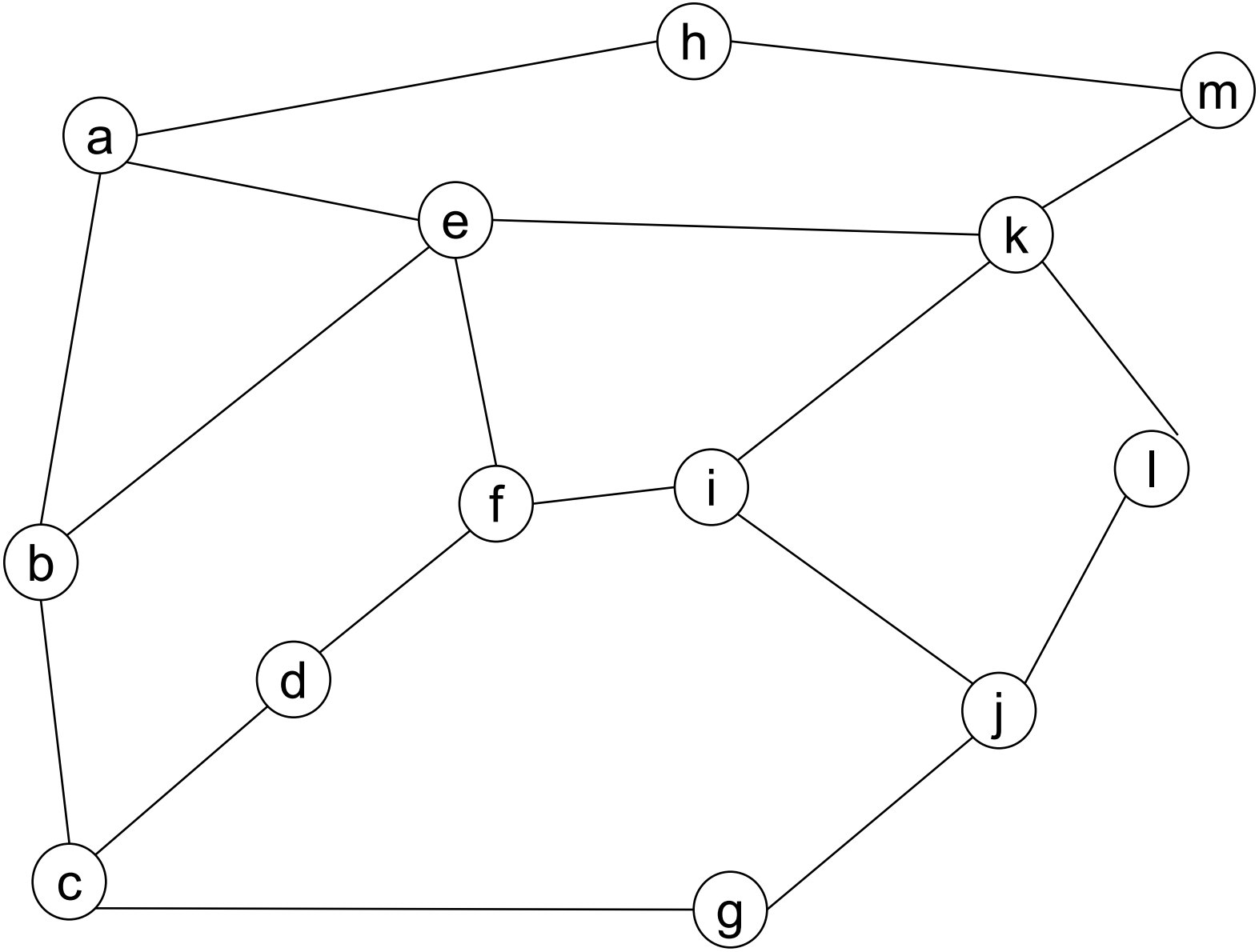



```
def preorder_print(node):
    if node:
        print(node.element)
        preorder_print(node.leftchild)
        preorder_print(node.rightchild)
```

recursive

```
def preorder_str(node):
    if node:
        outstr = '(' + str(node.element) + ' '
        outstr = outstr + preorder_str(node.leftchild)
        outstr = outstr + preorder_str(node.rightchild) + ')'
        return outstr
    else:
        return ''
```

(* (+ 4 2) (– 5 3))


h
m
a
e
k
l
f
i
b
d
j
c
g

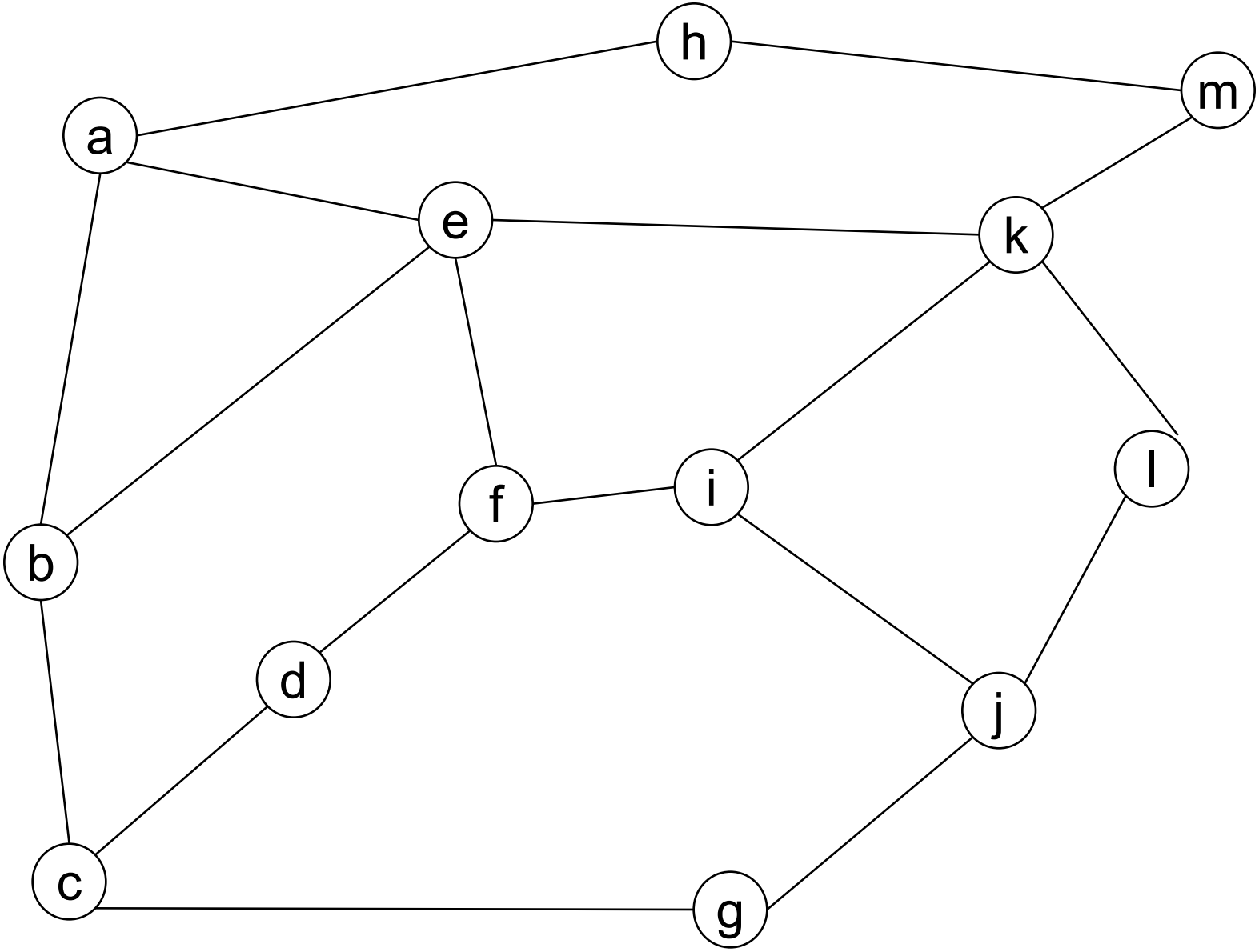
Depth-first search

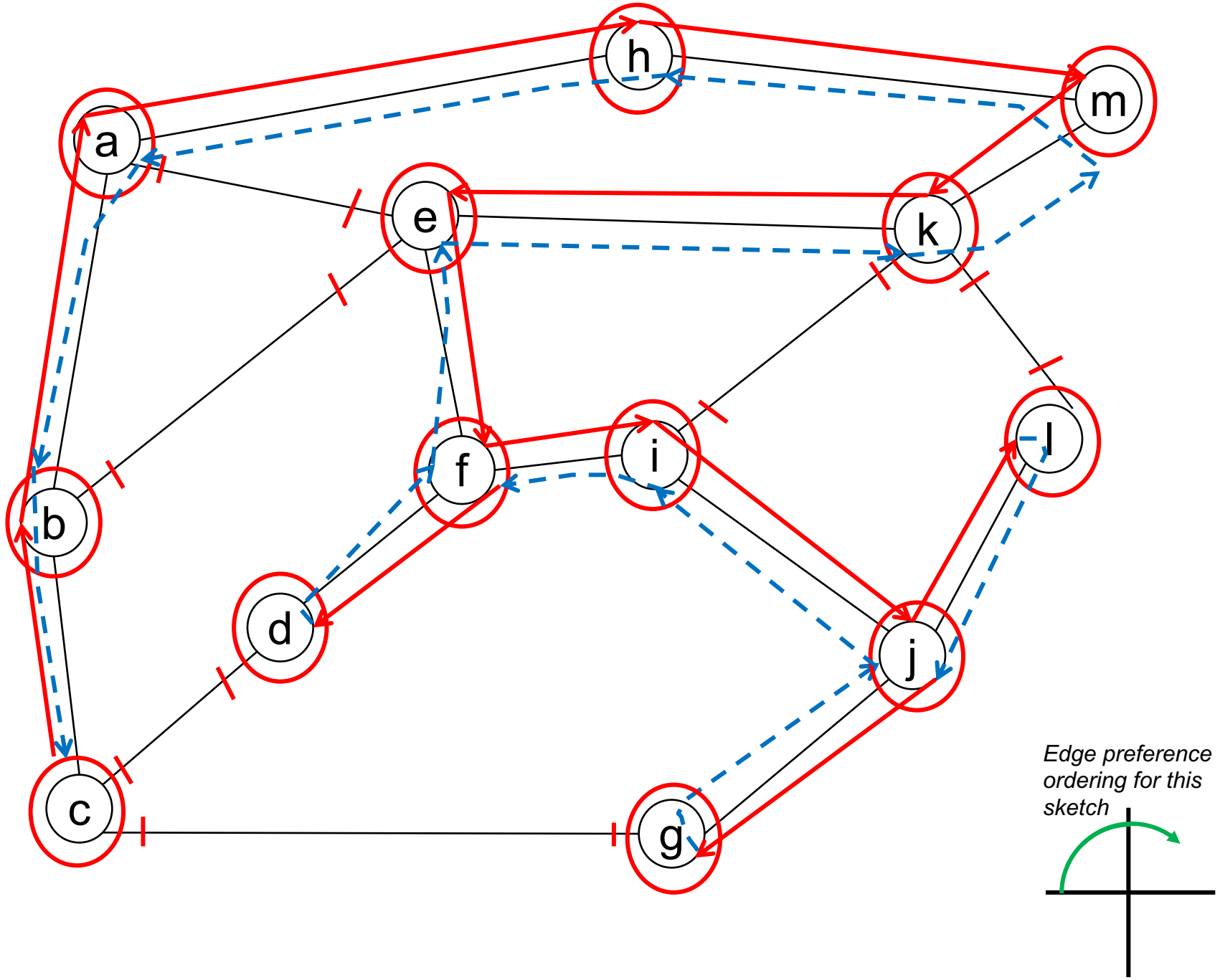
Think of exploring a maze, with a rope tied to the entrance, and with a can of paint.

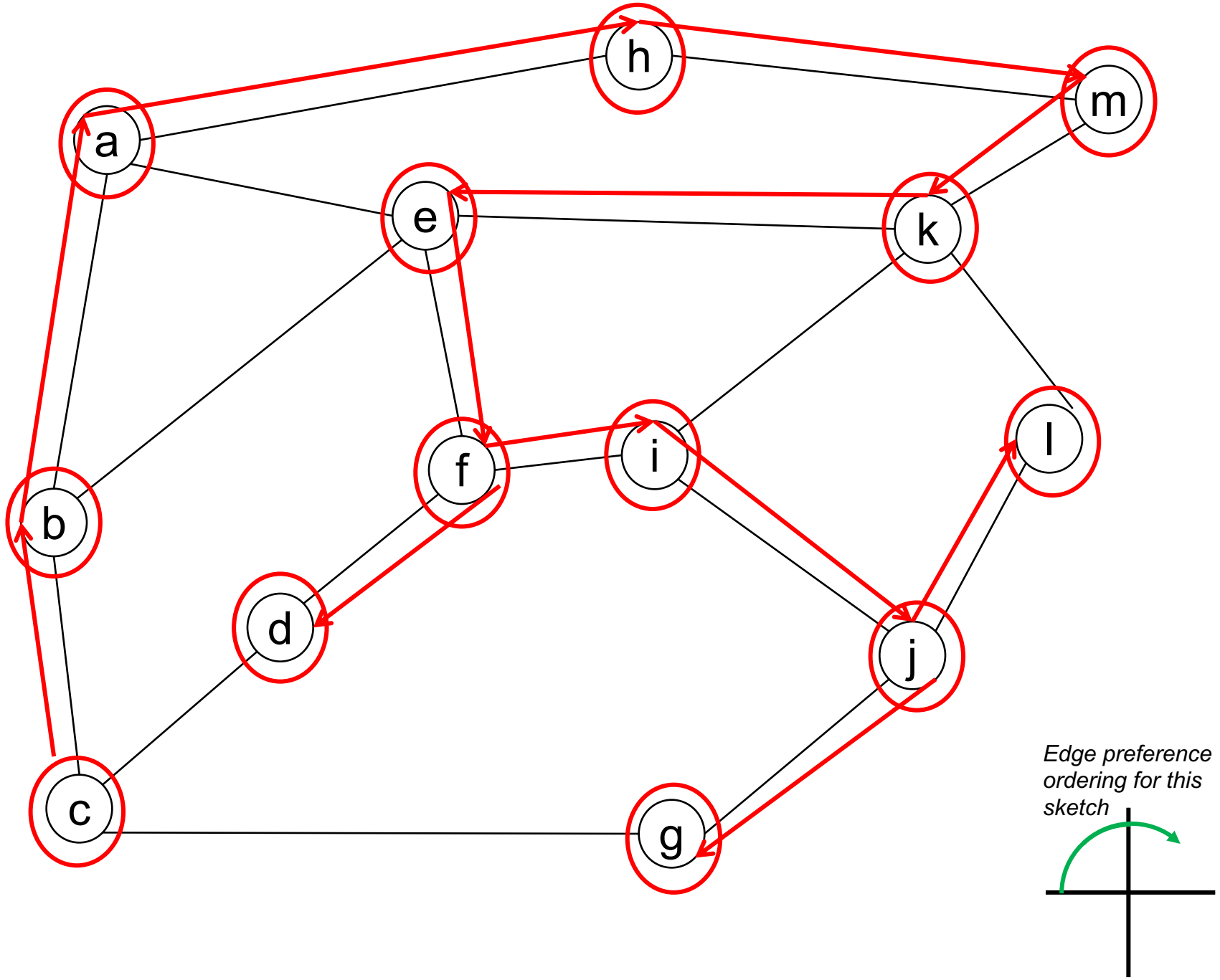
Mark the first vertex.
Choose an edge, and move along it to the opposite vertex; mark it.
Don't visit any vertex you have visited before.
When there are no unmarked vertices adjacent to the current one, backtrack to the vertex before the current one, and try the next edge from there.
When we are back at the first vertex, and there are no more edges we can try, stop.

Depth-first search: pseudocode

```
//pseudocode
depthfirstsearch(graph, v):
    mark v
    for each edge (v,w)
        if w has not been marked
            mark w
            depthfirstsearch(graph, w)
```


h
m
a
e
k
l
f
i
b
d
j
c
g


h
m
a
e
k
f
i
l
b
d
j
c
g
Edge preference ordering for this sketch


h
m
a
e
k
f
i
l
b
d
j
c
g
Edge preference ordering for this sketch

Properties

Depth first search (DFS):

1. the set of vertices marked in DFS is the *connected component* of G containing v

2. the set of marked vertices and the edges that led to them in DFS form a rooted spanning tree of the connected component, rooted at v

3. the worst case running time of DFS is O(n+m), where n is the number of vertices and m is the number of edges (depending on how we implement the algorithm ...)

The set of vertices marked in DFS is the connected component of G containing v

i.e. DFS from v reaches a vertex x if and only if there is a path from v to x

Proof?

Clearly, if DFS reaches x, then there must be a path from v to x, and so x is in v's connected component.

Suppose there is a vertex x in v's connected component, but DFS does not mark x.

Let P be a path from v to x (P must exist, since x is in v's connected component).

Let w be the first vertex in P that is not marked by DFS. (there must be one, and w might be x itself). Let y be the node before w in the path.

During DFS, when visiting y, we must have considered edge {y,w} and rejected it. But we only reject an edge if the vertex at the other end has been marked. Contradiction.
Therefore there cannot be a vertex like x, and so DFS marks every vertex in v's connected component.

The set of marked vertices and the edges that led to them in DFS form a rooted spanning tree of the connected component, rooted at v

Proof?

Each vertex in the connected component is marked.
The edges that lead to them trace out a path from the initial vertex v.
We never create a cycle, since we are checking for marked vertices.

A connected simple graph with no cycles is a tree, and this one has all edges oriented away from v

The tree contains every vertex in the connected component.

Therefore it is a spanning tree for the connected component.

The worst case running time of DFS is O(n+m), where n is the number of vertices and m is the number of edges
(assuming:

- we can obtain a list of edges incident on a vertex x in time O(degree(x))
- we can find the opposite vertex in an edge in O(1)
- we can mark a vertex, and test if it is marked in time O(1))

Proof?

There are n vertices in the graph, and m edges.

We call the DFS algorithm at most once on each vertex. We then find all edges for that vertex. There are m edges, and we consider each edge at most twice (once in the call by DFS for each of its two vertices). In the edge check, we find the opposite vertex, and check if it is marked.

So O(n+m).

Exercise: which of the Graph implementations support the three conditions above?

```
class Graph:

    def depthfirstsearch(self, v):
        marked = {v:None}
        self._depthfirstsearch(v, marked)
        return marked

    def _depthfirstsearch(self, v, marked):
        for e in self.get_edges(v):
            w = e.opposite(v)
            if w not in marked:
                marked[w] = e
                self._depthfirstsearch(w, marked)
```

Note: 'marked' is a dictionary, so lookup time is *expected* to be O(1). We could replace the dictionary with an array-based list of size n, and get a true O(1).

Exercises:

- for any vertex w, how would you reconstruct a path from v to w from the dictionary 'marked'?
- how would you generate a list of all connected components of a graph?
- how could you implement DFS without a recursive call?

Next lecture

Breadth first search

Directed graphs