Graph Traversals

Lecture 03.02

By Marina Barsky

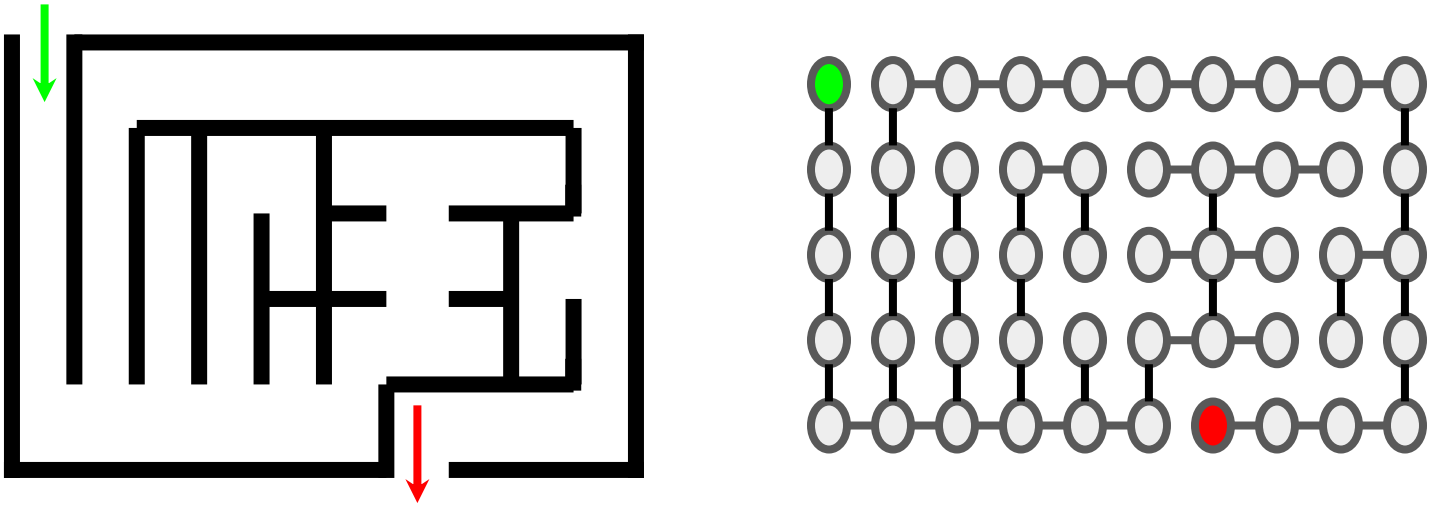
Graph Traversal

A graph traversal algorithm explores every vertex (and edge) of a graph.

It is one of the most important algorithms used as an archetype for solving many interesting problems:

- Courses and prerequisites (topological sort)
- Sub-networks (connected components)
- “Weak link” (articulation points)
- “Degree of separation” (shortest paths)
- …

Example: Mazes

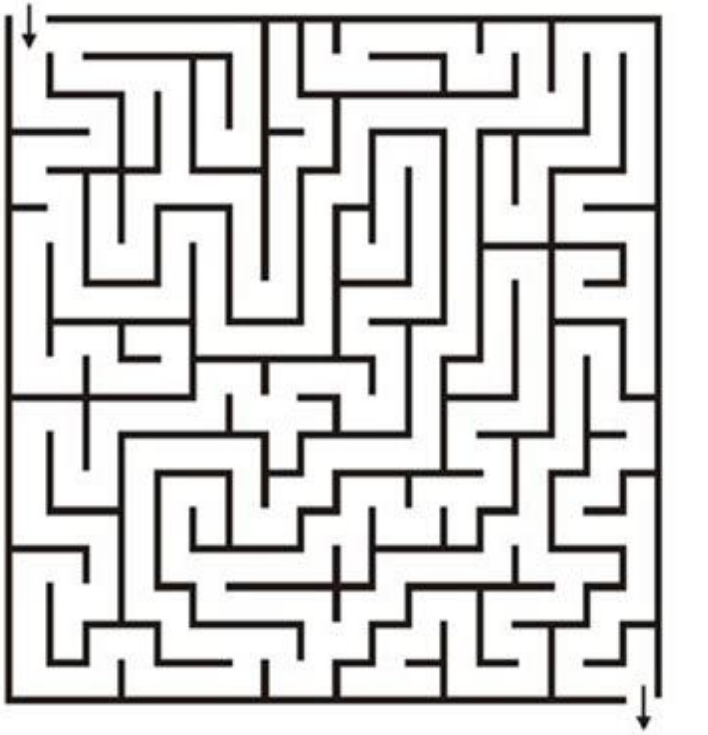


A maze can be viewed as a grid graph.

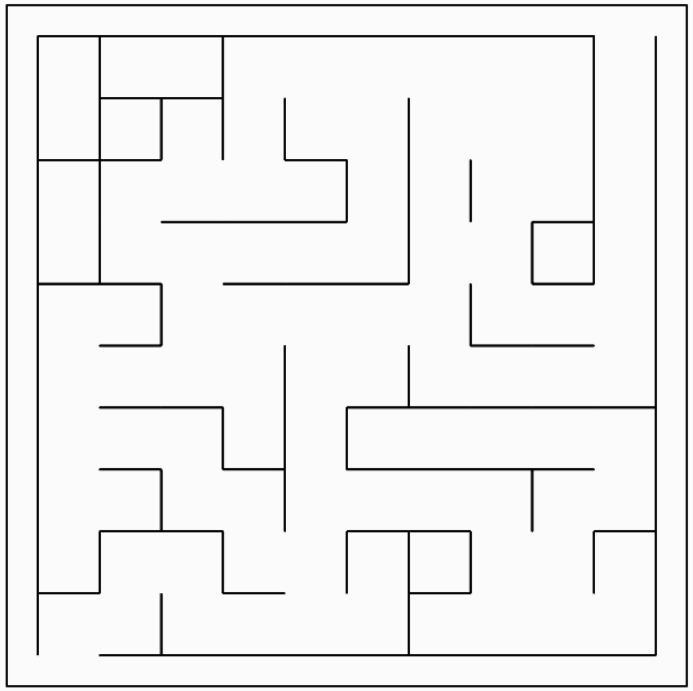
The vertices are pairs of (x,y) coordinates, and two neighboring cells are connected by an edge if there is no wall between them.

Perfect mazes

A paper-and-pencil maze that has no loops and no inaccessible areas is called a "perfect maze".



Perfect maze

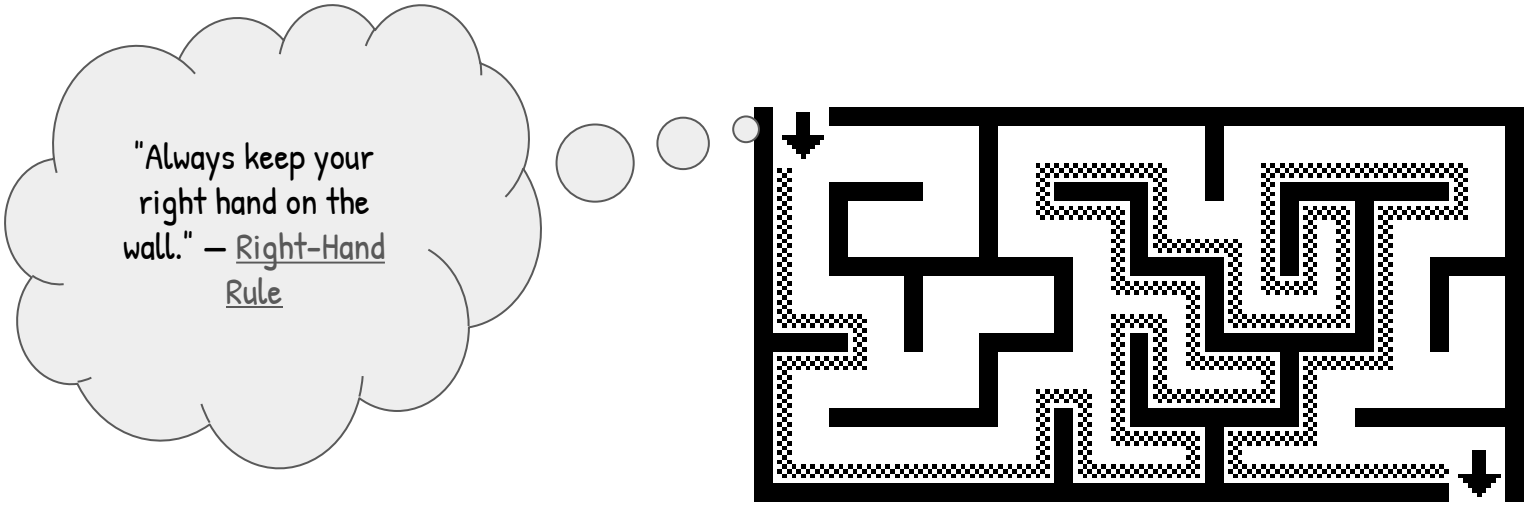


Imperfect maze

Solving perfect mazes

Perfect mazes can be solved using the right-hand rule.

(The left-hand rule also works and explores the maze in the opposite order.)

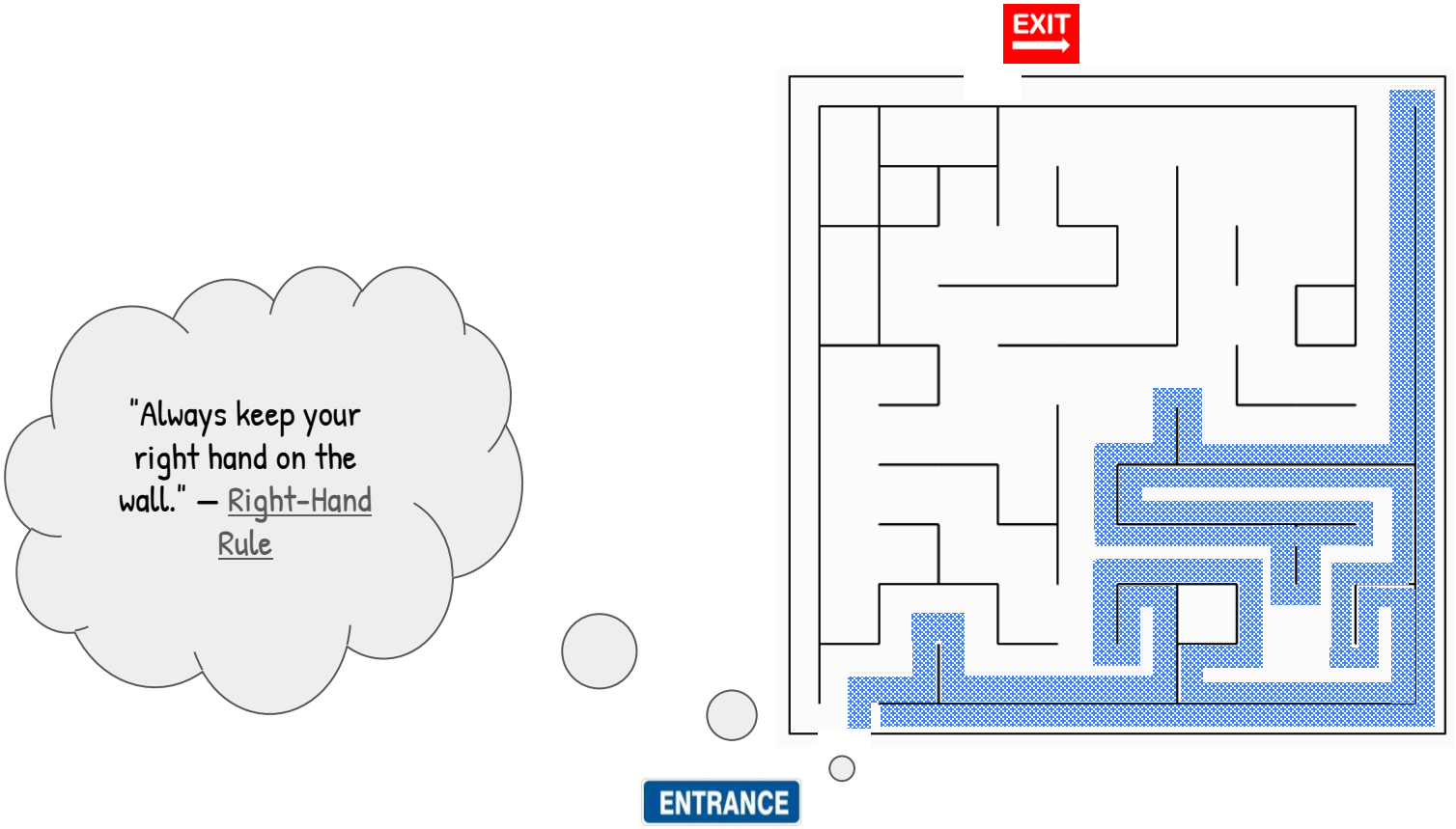


If you don't take the exit after you found it, then you will traverse the entire maze graph before returning to the entrance.

Solving perfect mazes

Perfect mazes are easier to traverse than arbitrary graphs since the underlying graph is acyclic.

The right-hand rule will not work for imperfect graphs or for arbitrary graphs.



Perspective: where to go next?

When solving paper mazes humans "see" the entire maze all at once and can use their intuition.

Computer programs work one step at a time and only "see" one thing.

In graph traversal algorithms we need to use a first-person perspective.

Types of Graph Traversal

Recall tree traversal algorithms:

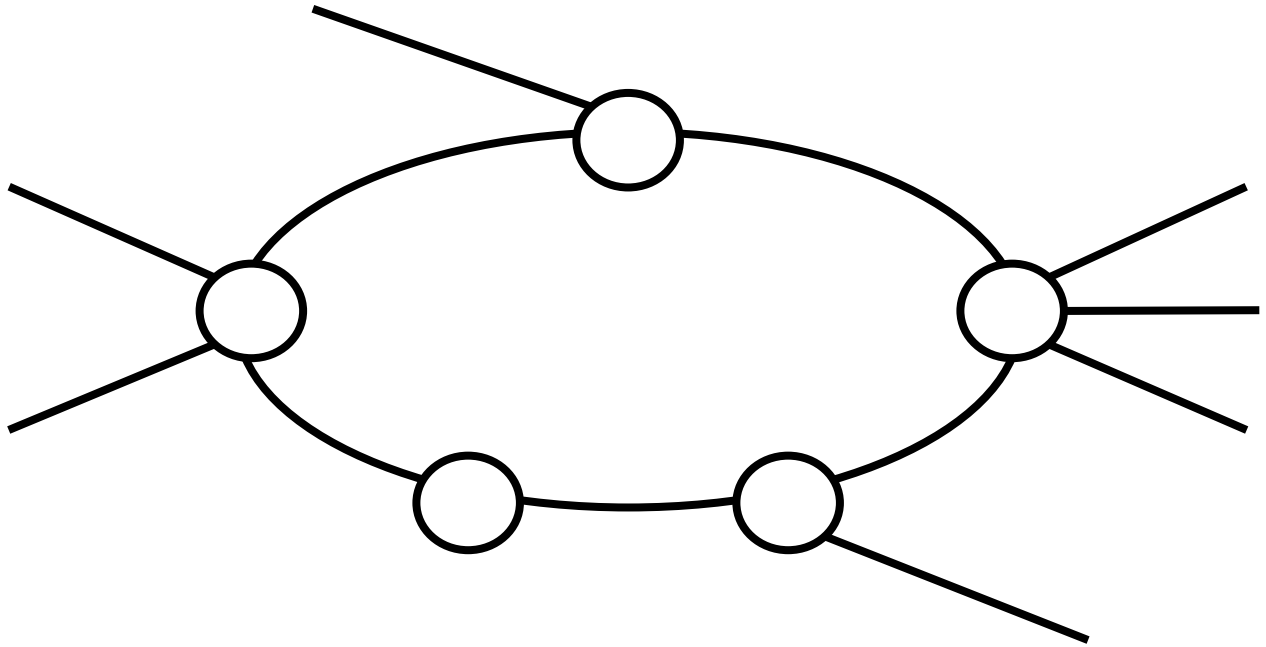
- Depth First:
  - In-order
  - Pre-order
  - Post-order
- Breadth First

Traversing general graphs is similar:

- Breadth-First Search
- Depth-First Search

When traversing general graphs (as opposed to trees) we need to be more careful.

We may return to the same node and loop around it endlessly.



We may end up looping around this cycle.

Endless Looping

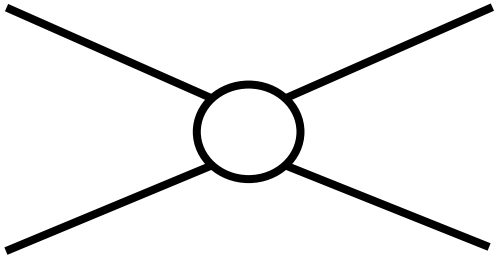
Super 3D Noah's Ark

We try to avoid doing this!

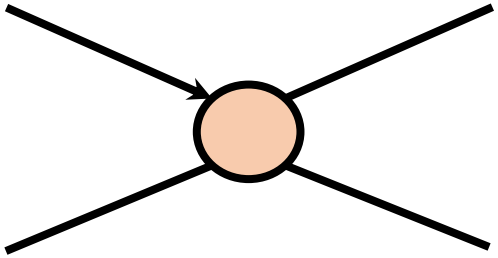
Labeling Vertices

To avoid the looping problem we will keep track of some data.

Each vertex will be marked as being in one of three different states.

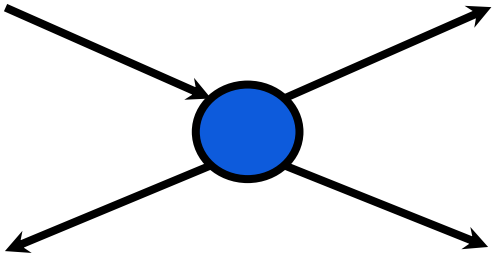


The vertex has not been seen by the algorithm yet.



The vertex has been seen, but we haven't fully explored its edges and adjacent vertices yet.

It has **not** been processed.

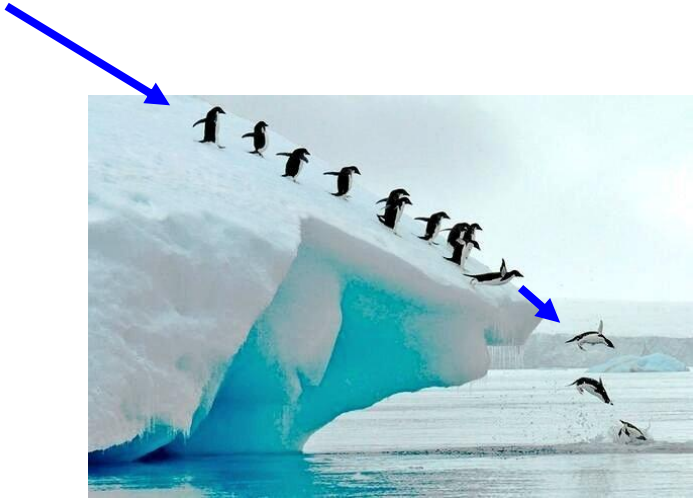


The vertex has been seen, and we have fully explored its edges and adjacent vertices.

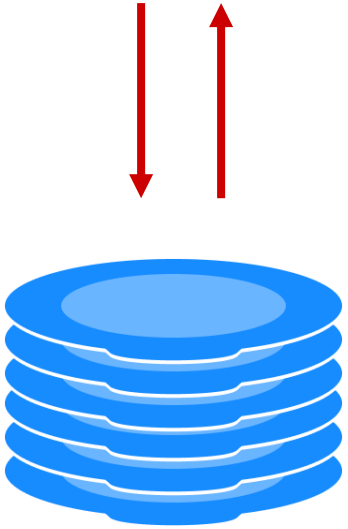
It has been processed.

1. A vertex is ***undiscovered*** if the algorithm has not seen the vertex yet.
2. A vertex is ***discovered*** if the algorithm has seen the vertex but we have not followed all of its incident edges.
3. A vertex is ***processed*** if the algorithm has seen the vertex and followed all of its incident edges.

We would also need: Queues and Stacks



Queue

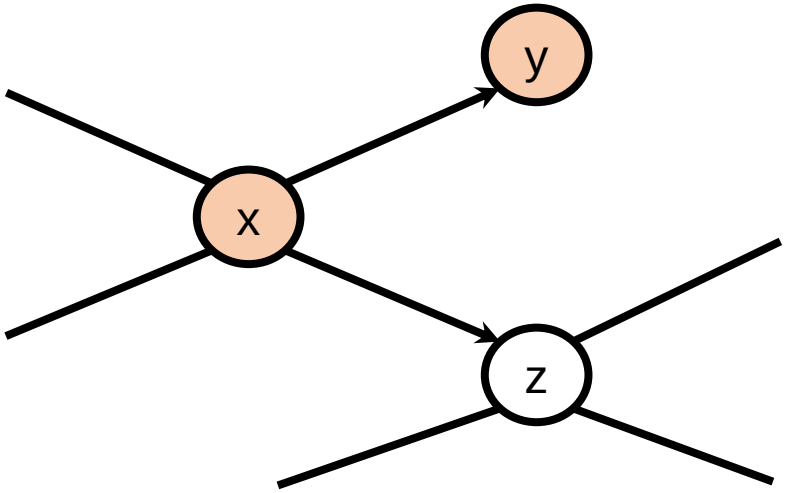


Stack

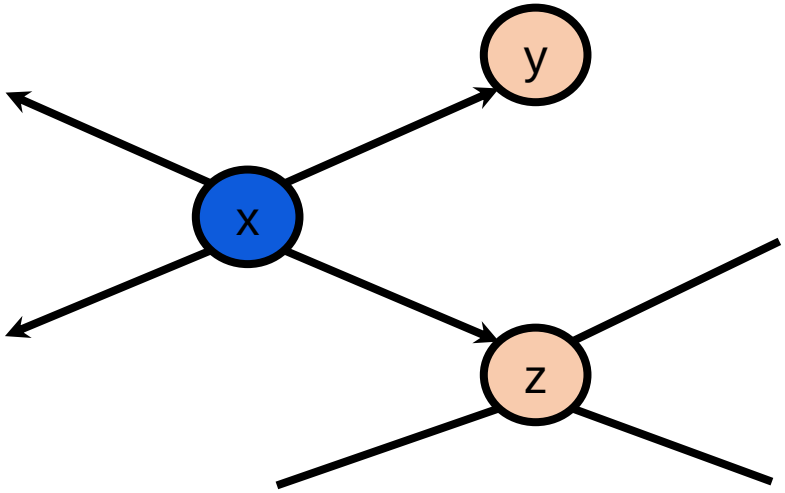
To-Do List

In a special To-Do list we will keep track of vertices that have been discovered but not yet processed.

Whenever we encounter an undiscovered vertex we add it to a *to-do list*.



While processing vertex x we discover vertex z, and we rediscover vertex y.



We have previously discovered y, so it is already in the to do list (or it has already been fully processed).

We have not previously discovered z so it is added to the to-do list.

Our algorithms repeatedly examine vertices in the to-do list until there are none.

- Initially the to-do list contains the single vertex where we start.
- For consistency of the discussion, when processing we examine a vertex's neighbors clockwise from 12 o'clock.

Generic Algorithm for Graph Traversal

The following algorithm will correctly and efficiently explore all the vertices.

Algorithm traverse (graph G, vertex start)

```
1    for each u in vertices of G:
2        u.state:= "undiscovered"

3    start.state:= "discovered"
4    todo:= new to_do_list()
5    todo.add(start)

6    while todo is not empty:
7        current_vertex:= todo.remove()
8        for each u in neighbors(current_vertex):
9                if u.state = "undiscovered":
10                       u.state:= "discovered"
11                       todo.add(u)

12       current_vertex.state:= "processed"
```

Breadth-First Search

In breadth-first search (**BFS**) the to-do list is a (FIFO) queue.

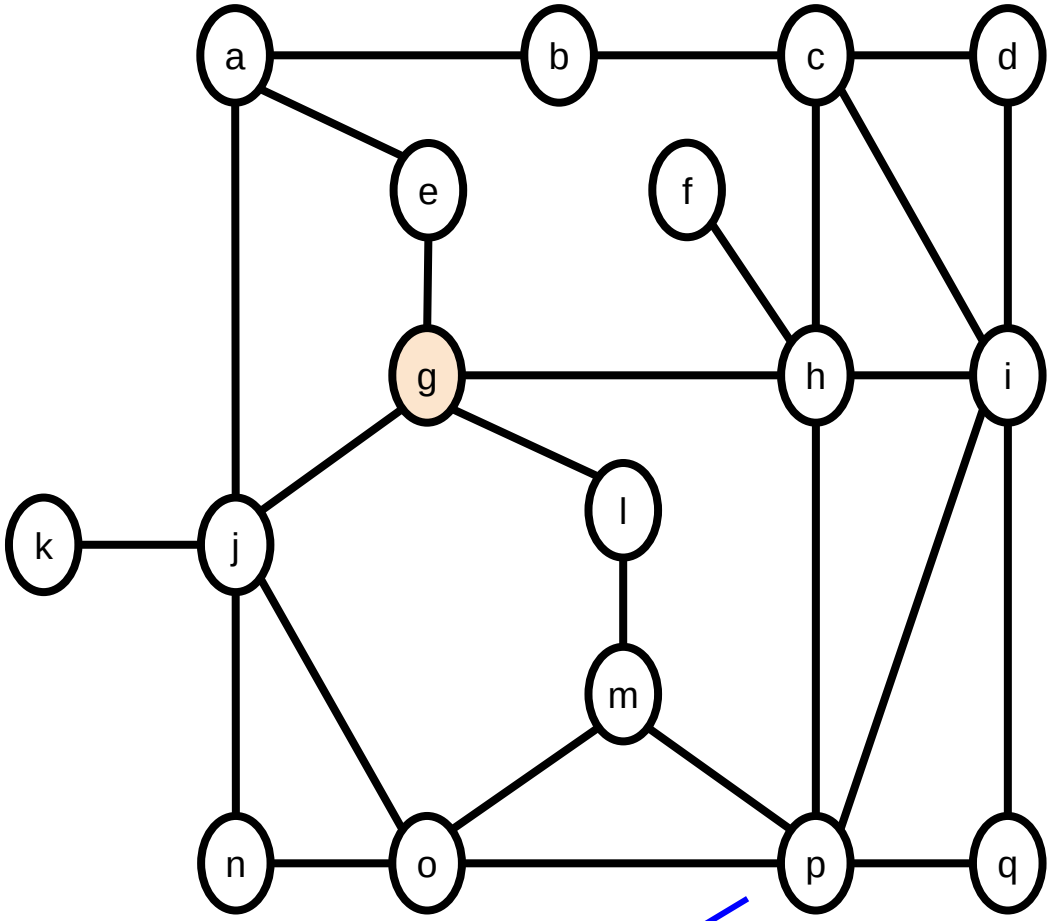
Algorithm BFS (graph G, vertex start)

```
1   for each u in vertices of G:
2      u.state:= "undiscovered"

3   start.state:= "discovered"
4   todo:= new Queue()
5   todo.enqueue(start)

6   while todo is not empty:
7      current_vertex:= todo.dequeue()
8      for each u in neighbors(current_vertex):
9            if u.state = "undiscovered":
10                u.state:= "discovered"
11                todo.enqueue(u)

12     current_vertex.state:= "processed"
```

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



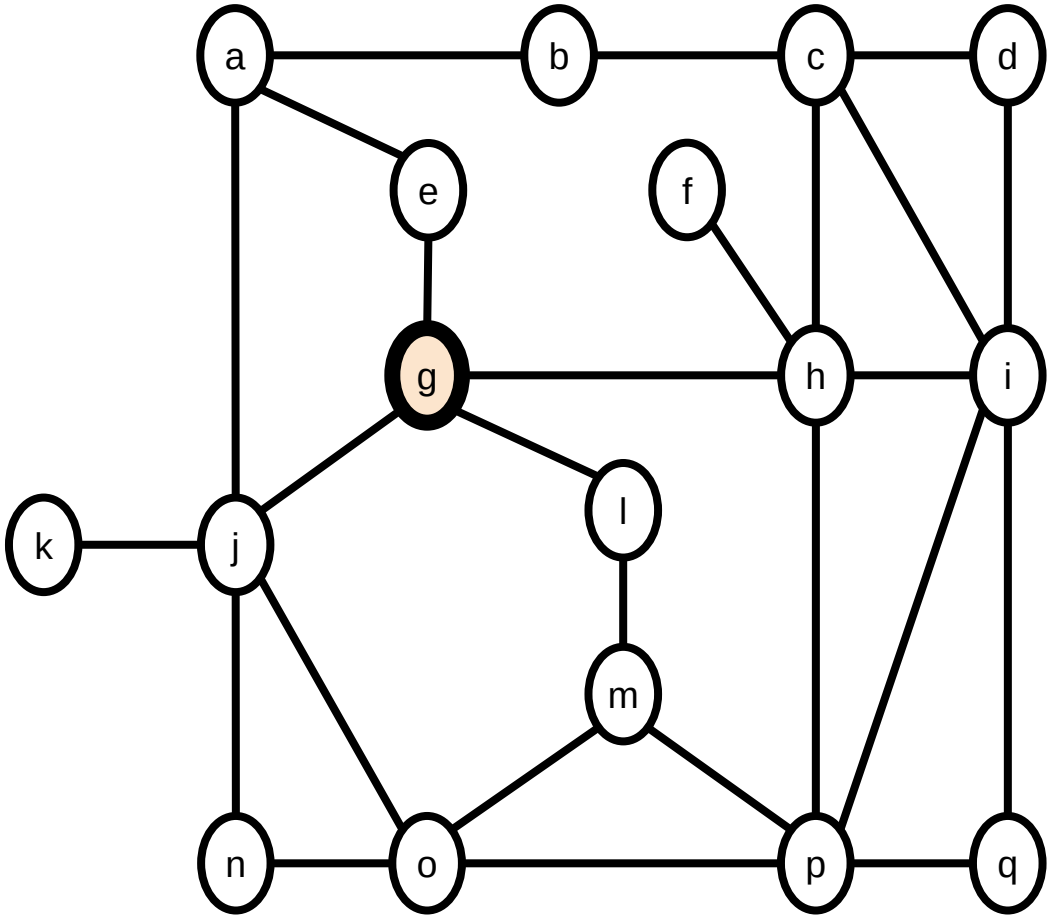
current_vertex

todo | g | | | | | | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

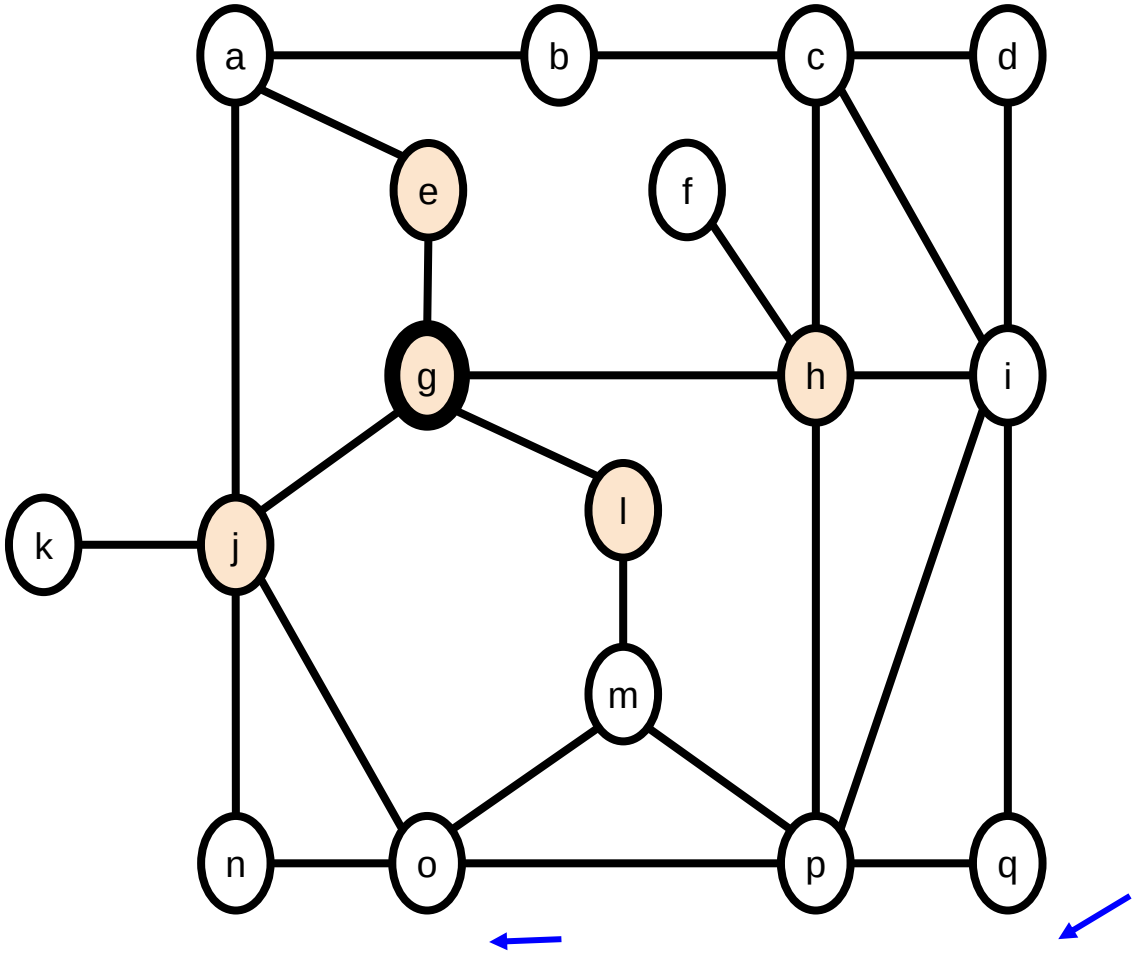


current_vertex | g | todo | | | | | | | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



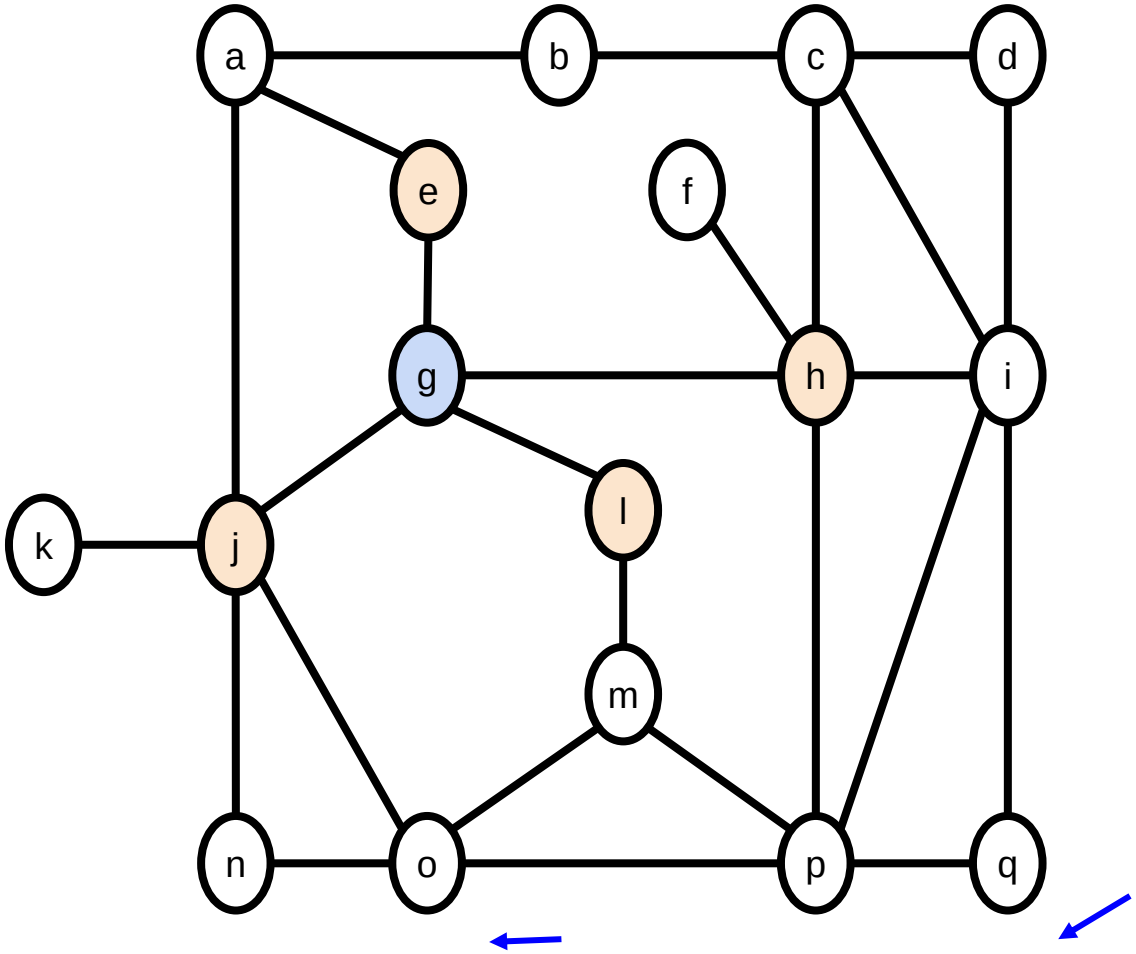
current_vertex | g |

todo | e | h | l | j | | | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



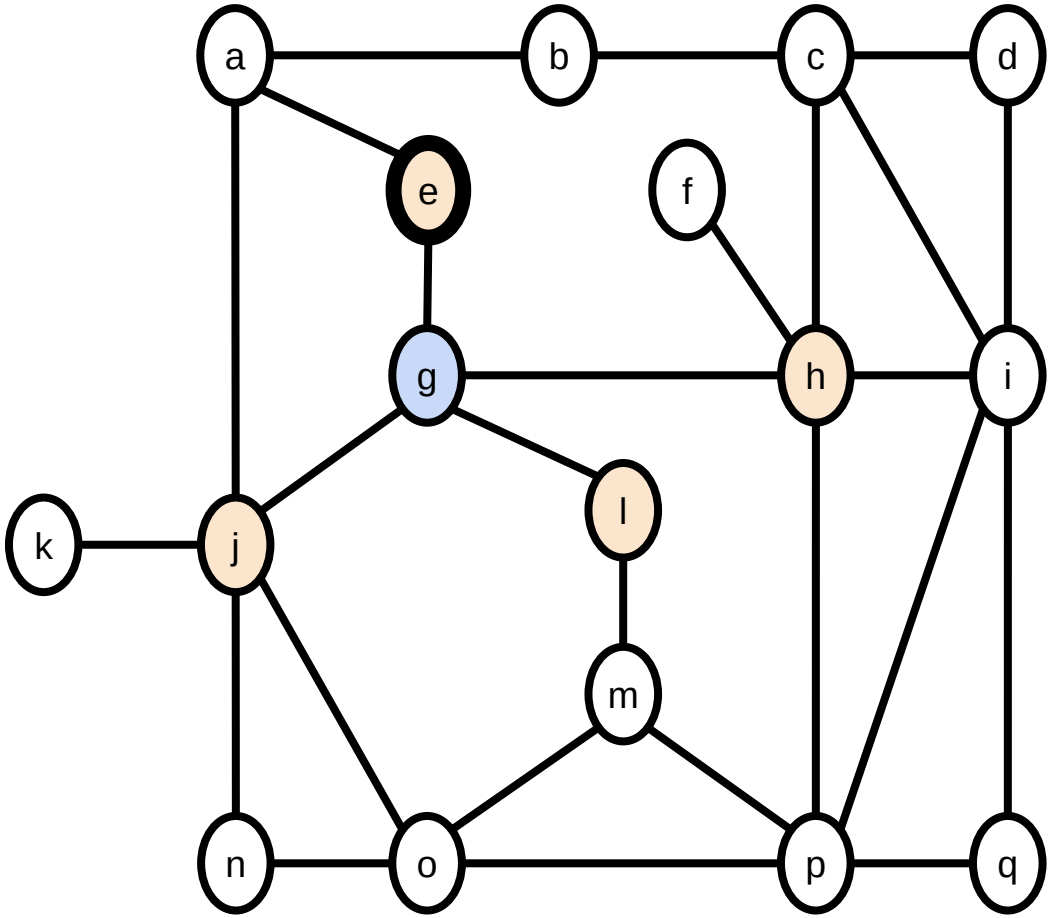
current_vertex

todo | e | h | l | j | | | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



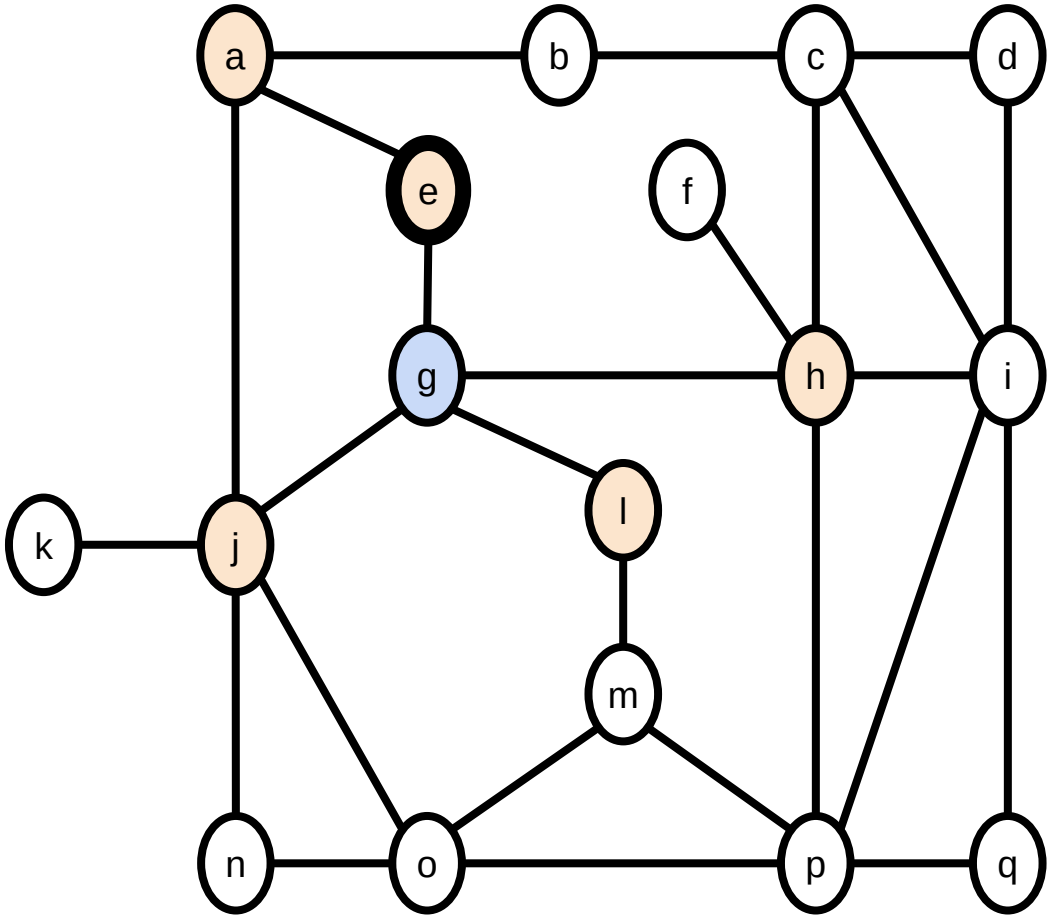
current_vertex | e |

todo | h | l | j | | | | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



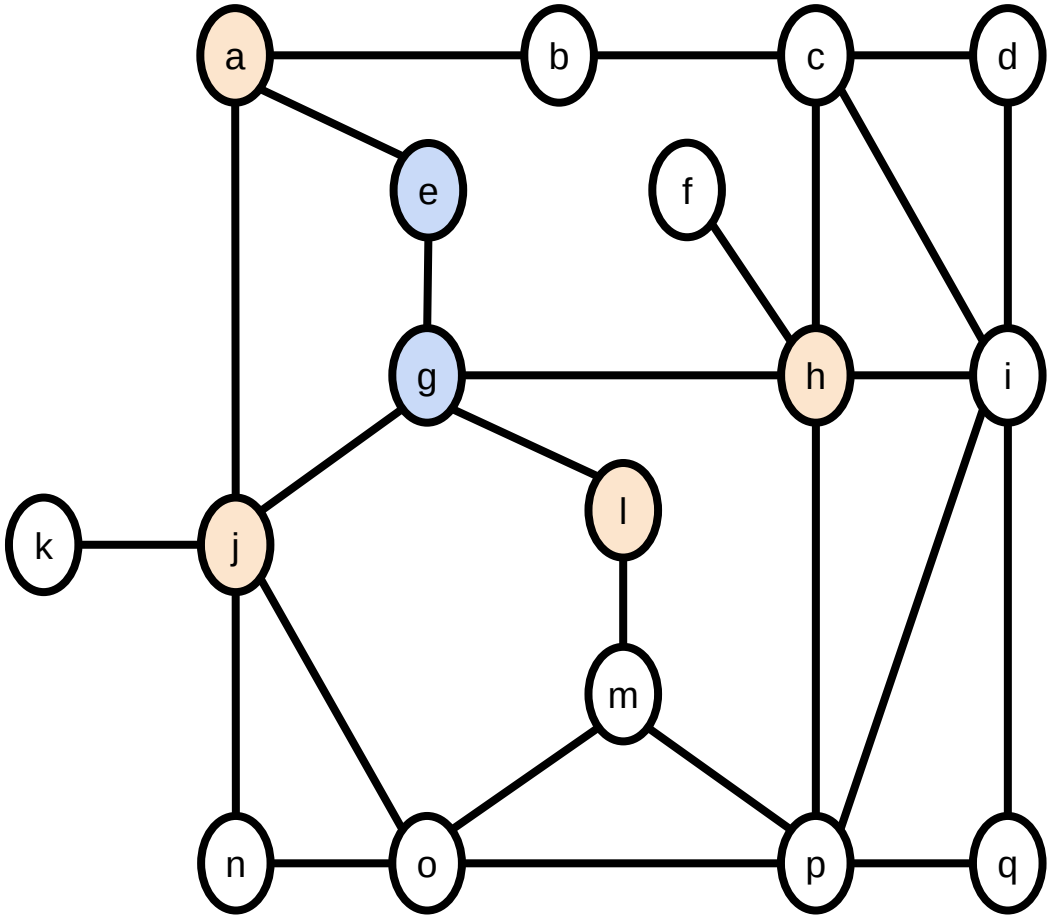
current_vertex | e |

todo | h | l | j | a | | | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



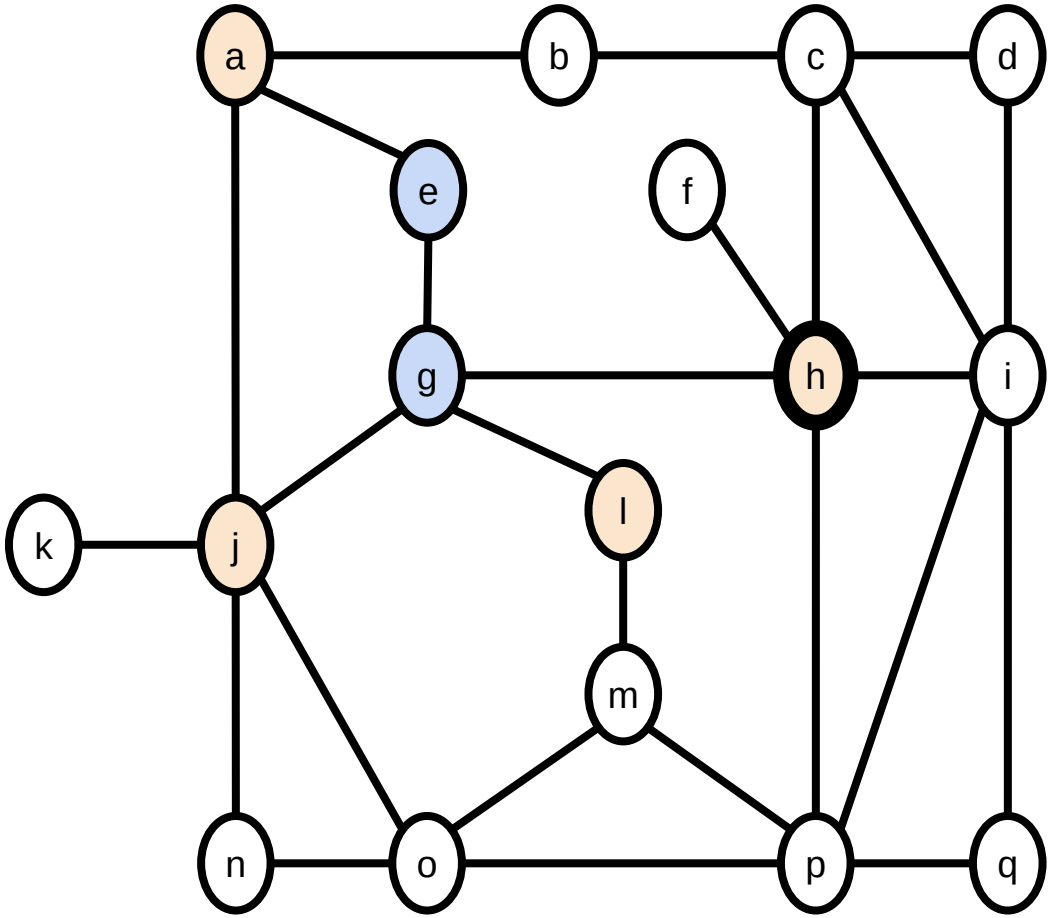
current_vertex

todo | h | l | j | a | | | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



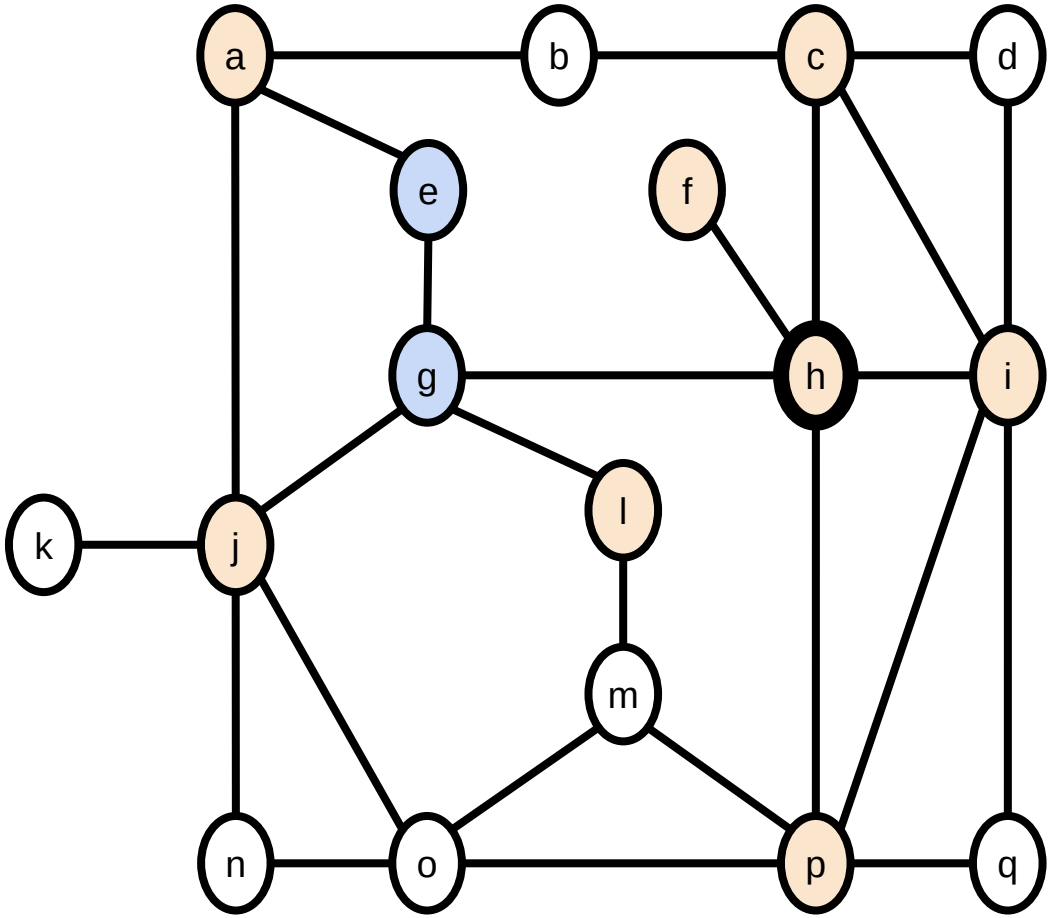
current_vertex | h |

todo | l | j | a | | | | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



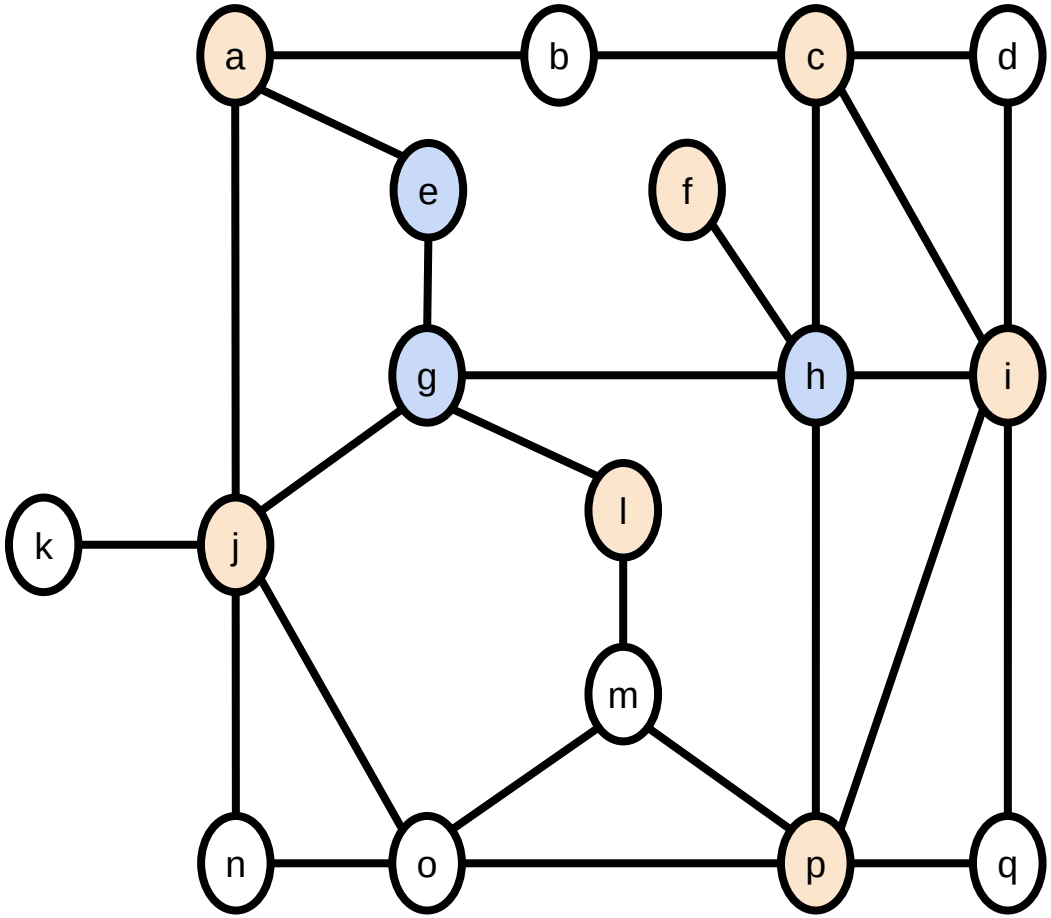
current_vertex | h |

todo | l | j | a | c | i | p | f | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



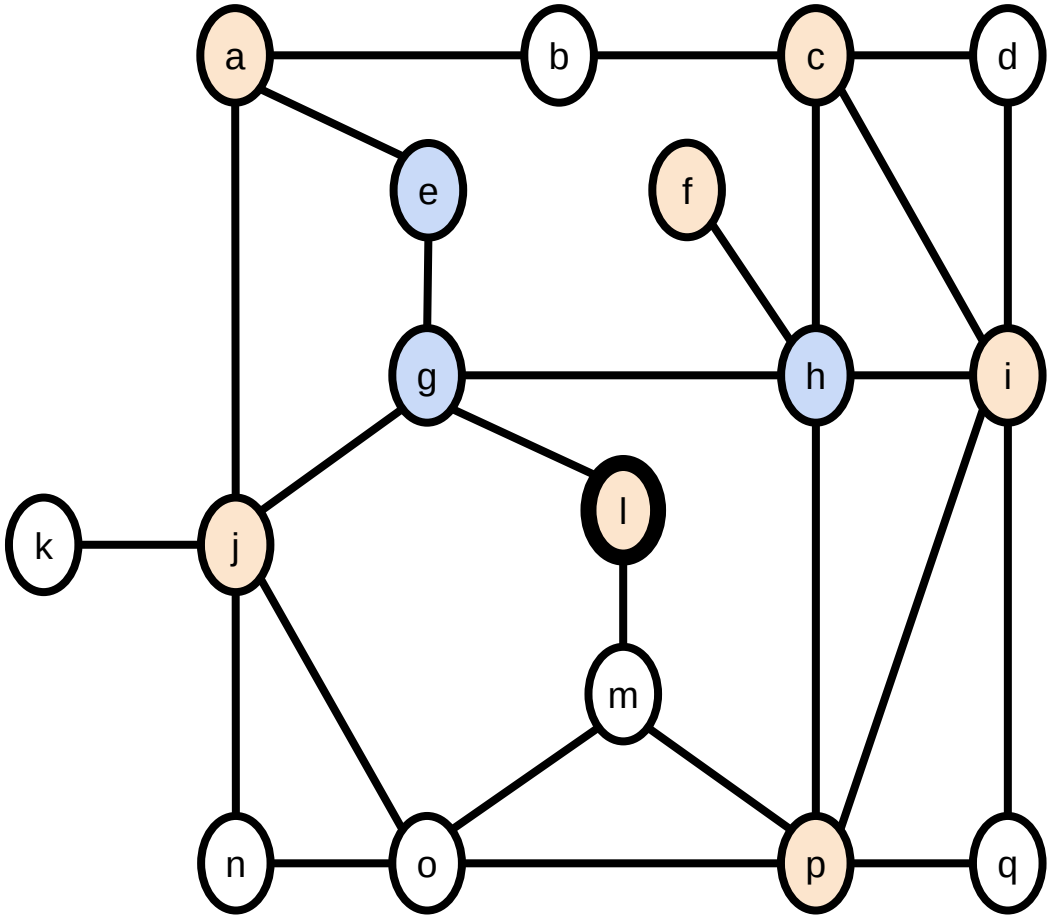
current_vertex

todo | l | j | a | c | i | p | f |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



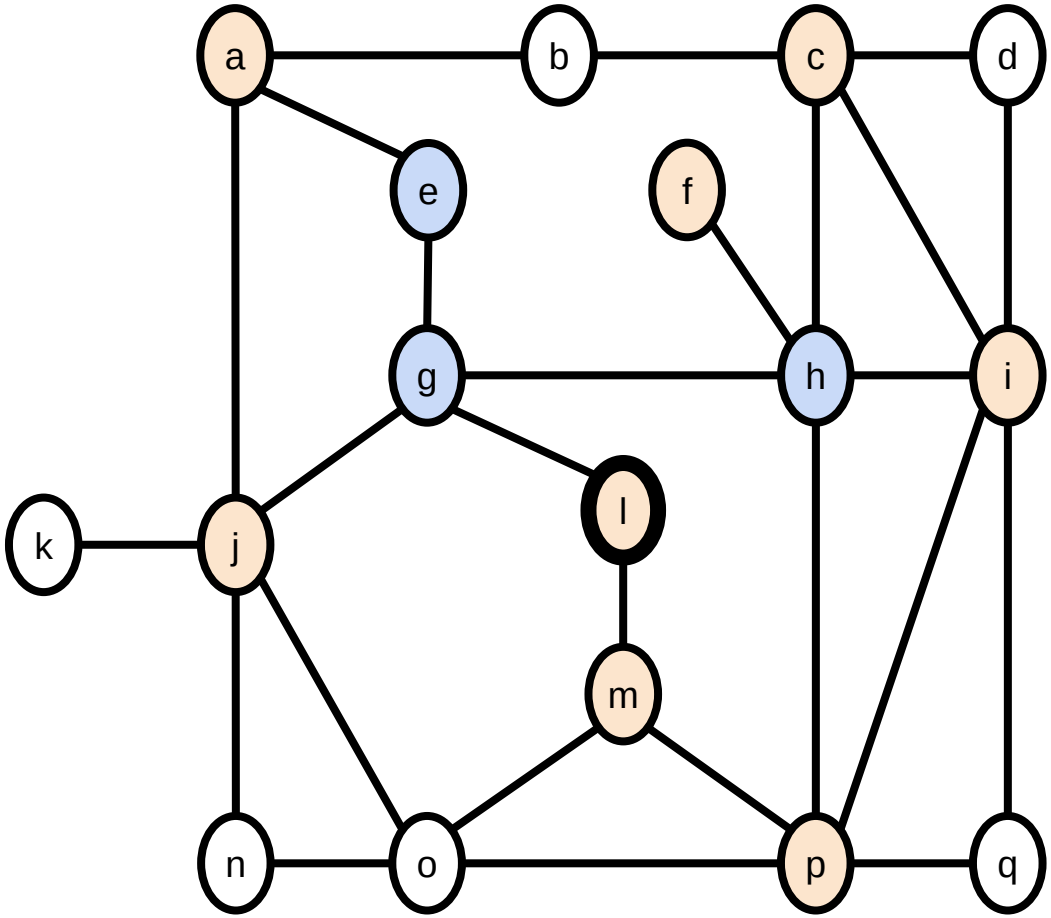
current_vertex | l |

todo | j | a | c | i | p | f | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



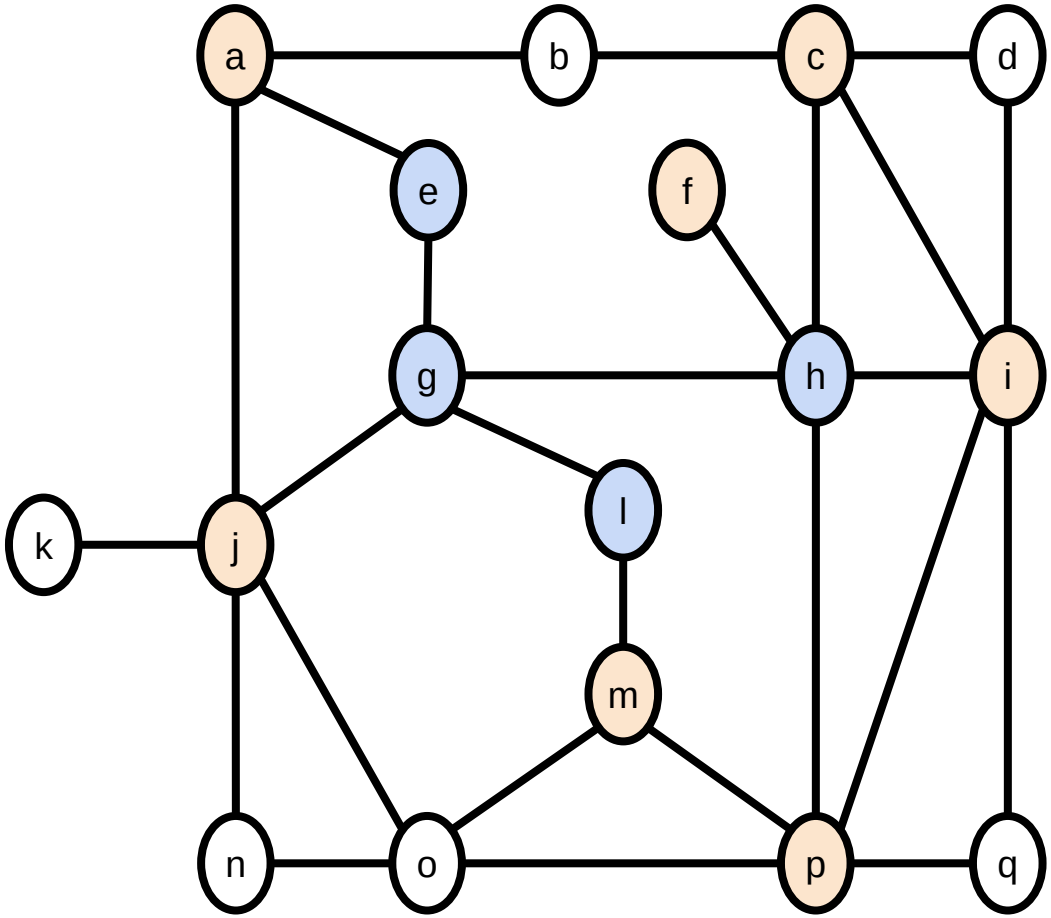
current_vertex | l |

todo | j | a | c | i | p | f | m | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



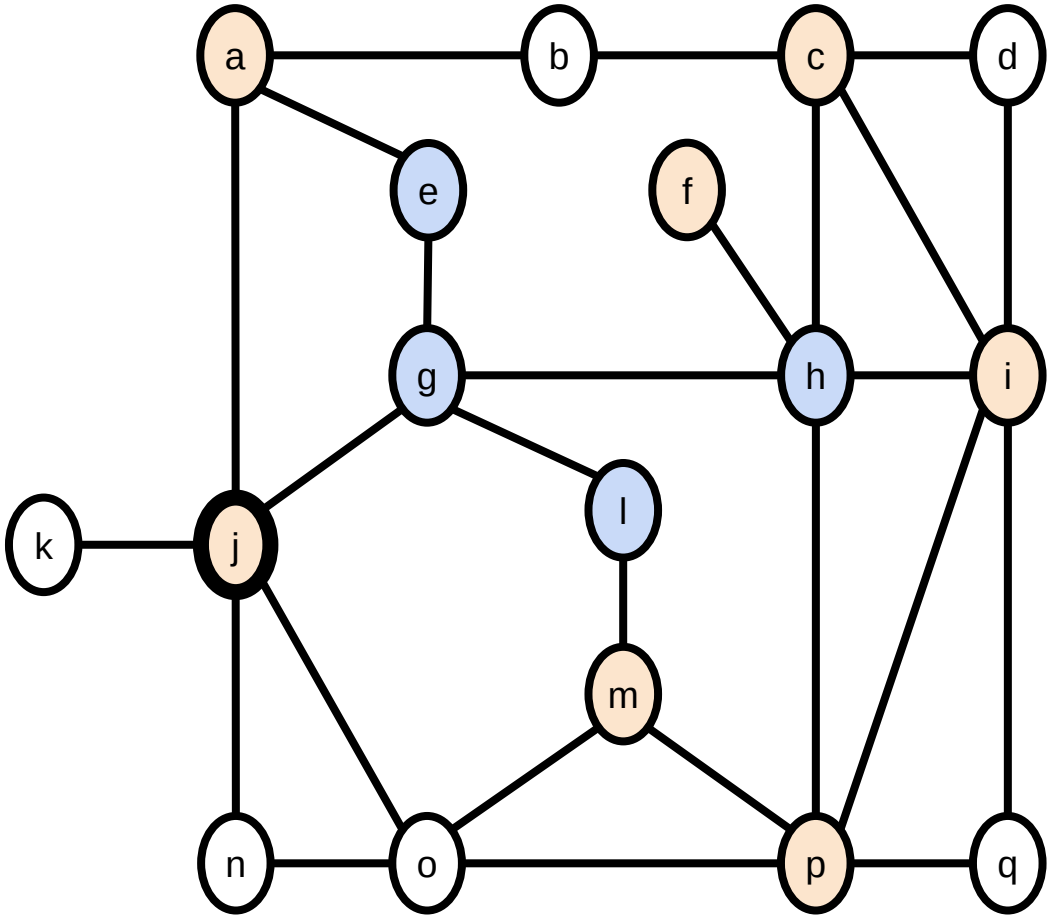
current_vertex

todo | j | a | c | i | p | f | m |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



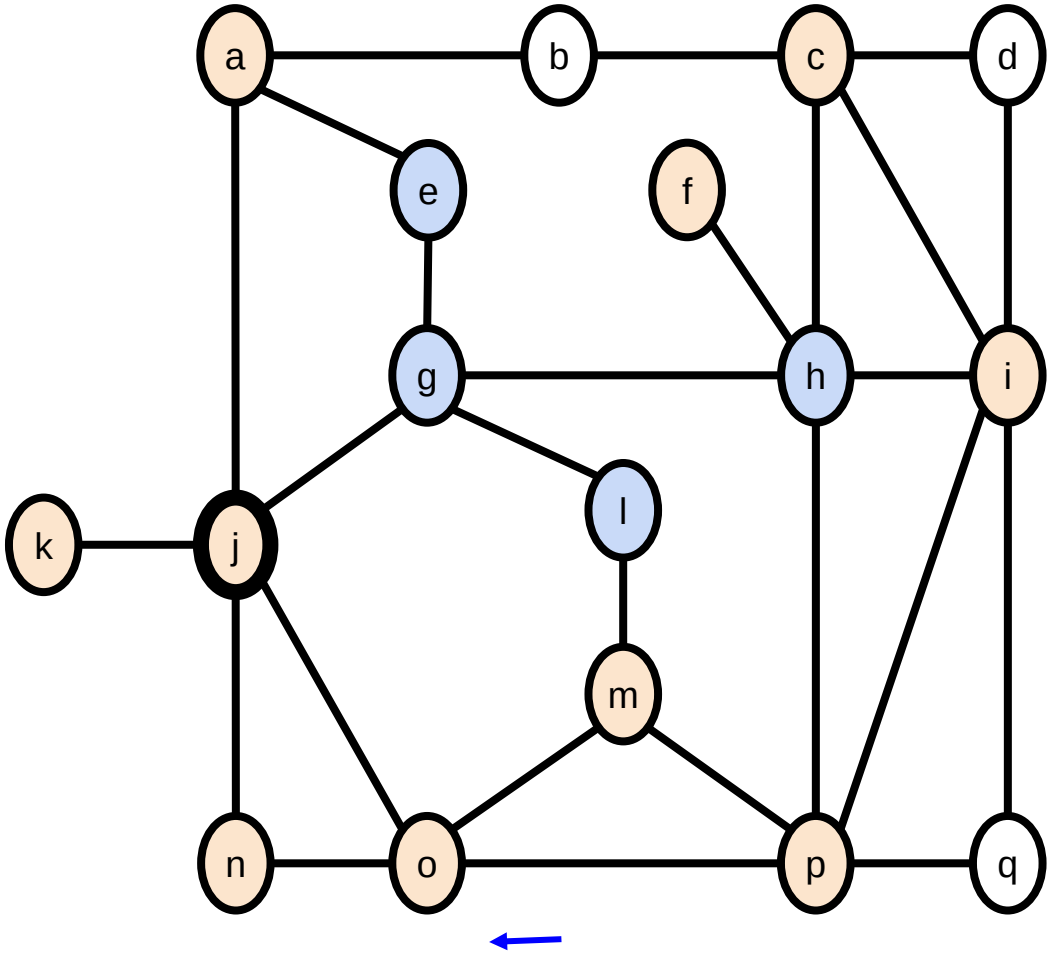
current_vertex | j |

todo | a | c | i | p | f | m | | |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



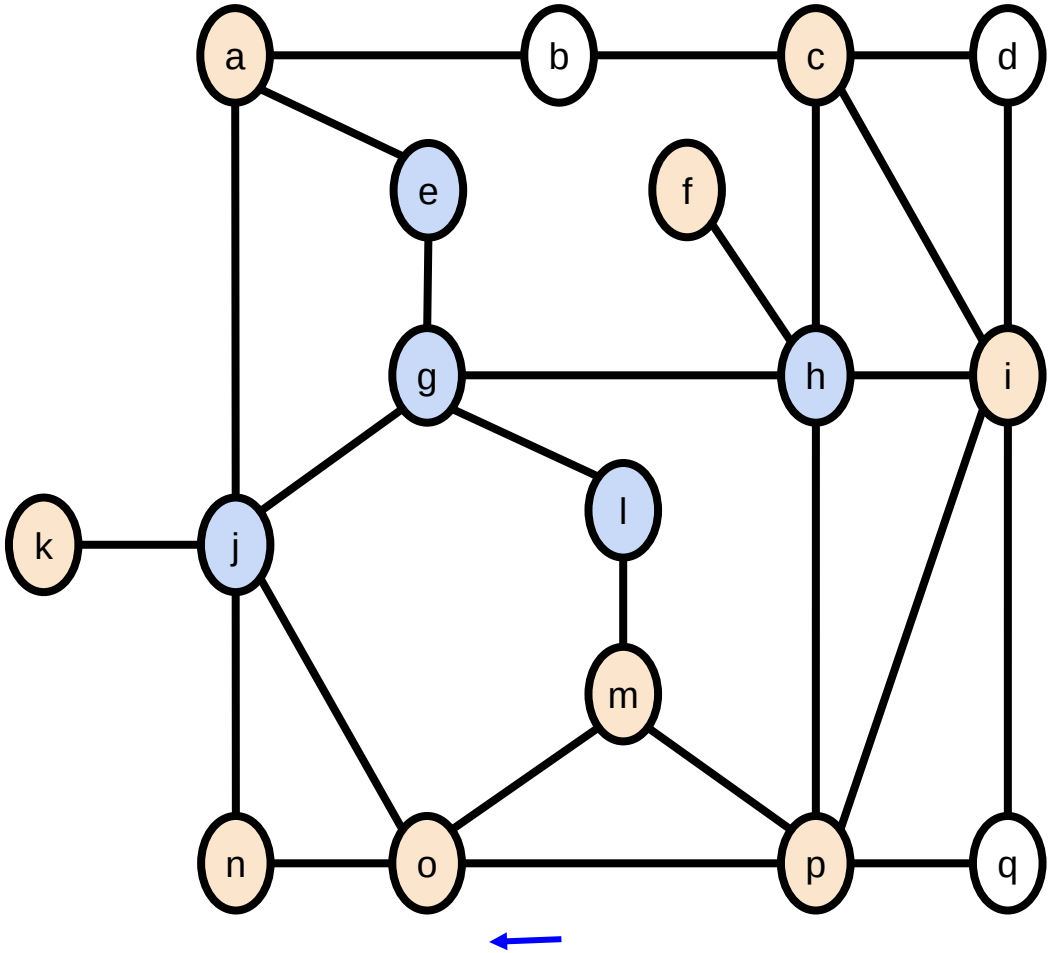
current_vertex | j |

todo | a | c | i | p | f | m | o | ... |

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

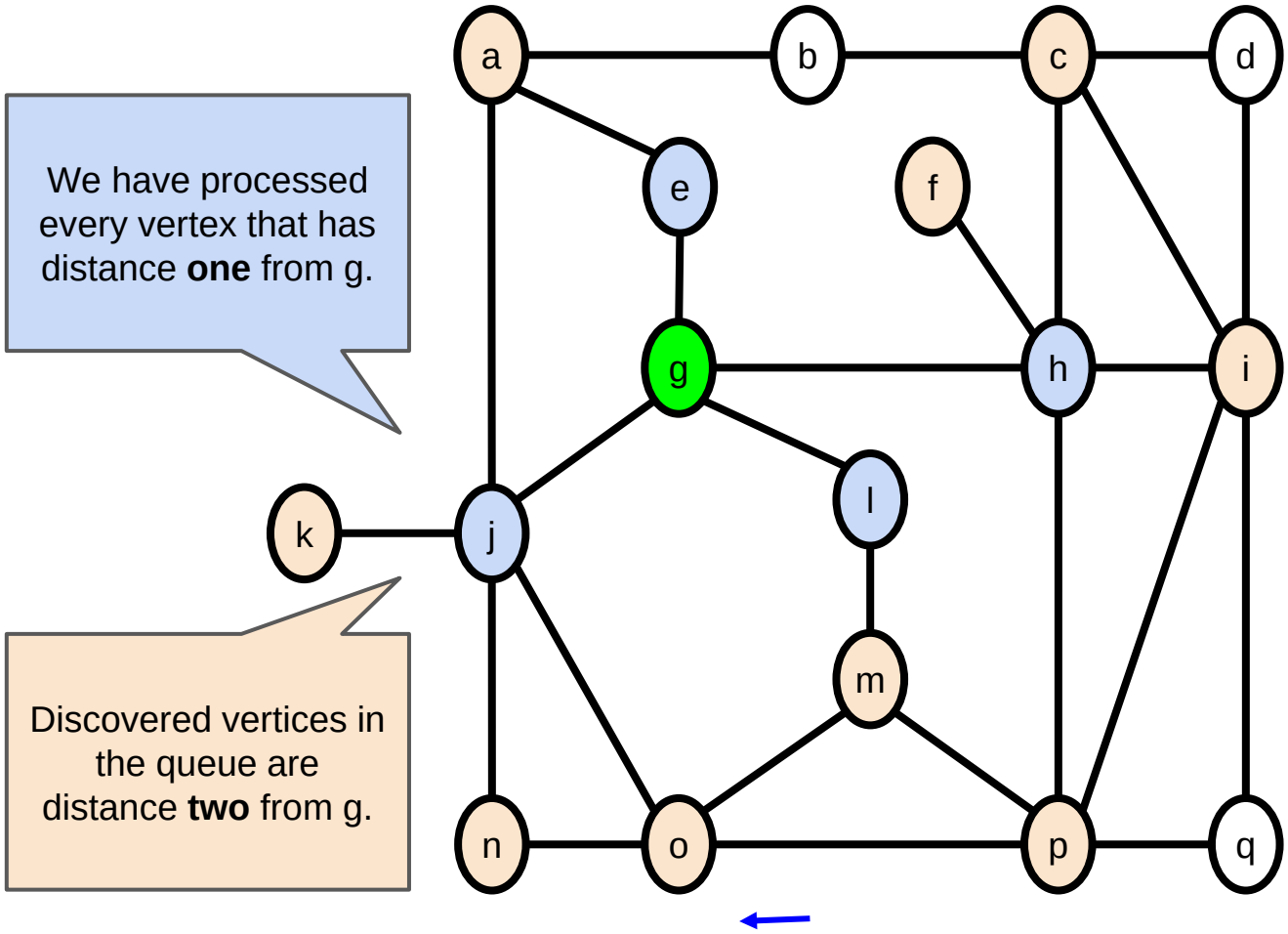


current_vertex | | todo | a | c | i | p | f | m | o | ...

Example: BFS

Breadth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



current_vertex [ ]

todo [a | c | i | p | f | m | o | ...]

BFS

1. Breadth-first search will always visit closer nodes first. In other words, a node at distance *d* edges from the start will be visited before any node at distance *d*+1.

2. The order that the nodes are discovered is the same order that the nodes are processed. Both of these orders can be obtained from the todo list by crossing out the values instead of erasing them.

Note: On the exam you may be asked to specify the order that the nodes are discovered / processed. For the previous example the answer would be:

g	e	h	l	j	a	c	i	p	f	m	o	n	k	b	d	q

Depth-First Search

In depth-first search (**DFS**) we process the to-do list as a stack.

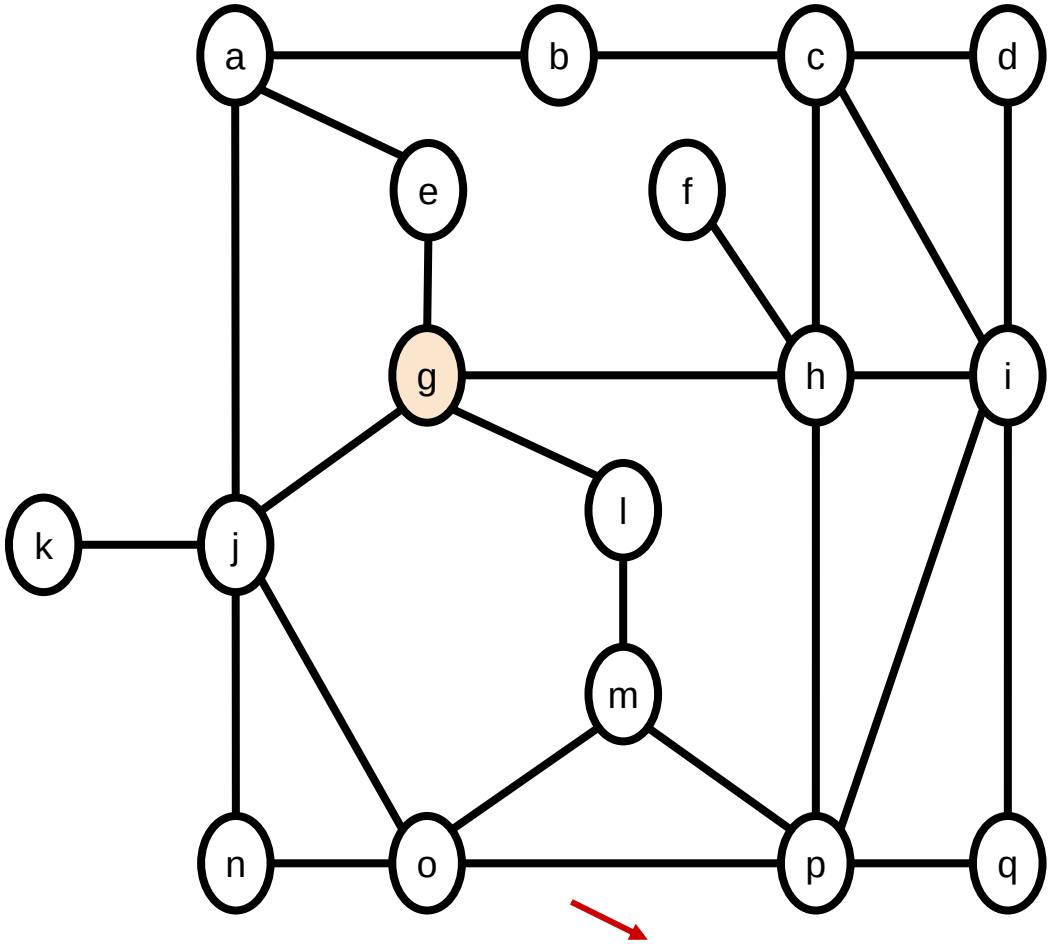
Algorithm DFS (graph G, start)

```
1   for each u in vertices of G:
2       u.state:= “undiscovered”

3   start.state:= “discovered”
4   todo:= new stack()
5   todo.push(start)

6   while todo is not empty:
7       current_vertex:= todo.pop()
8       for each u in neighbors(current_vertex):
9             if u.state = “undiscovered”:
10                  u.state:= “discovered”
11                  todo.push(u)

12      current_vertex.state = “processed”
```

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



current_vertex

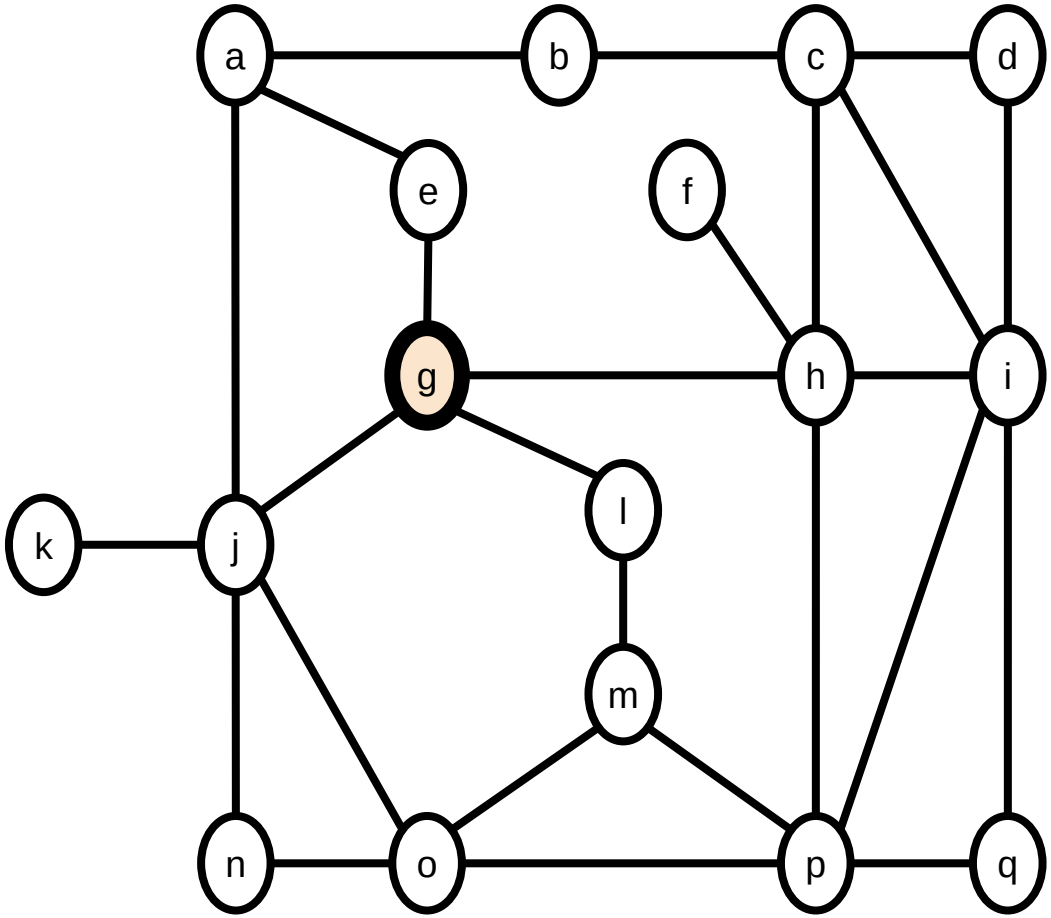
todo

g

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

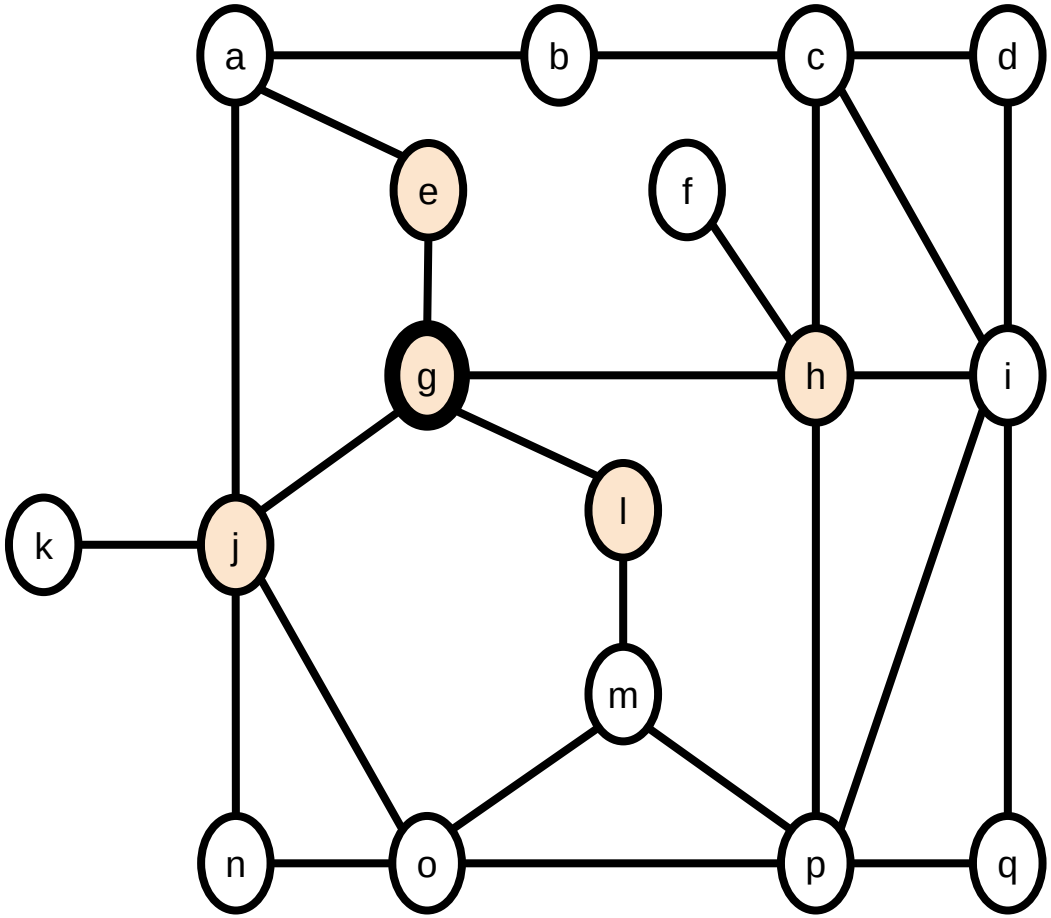


current_vertex | g | todo

Example: DFS

color	state
	undiscovered
	discovered
	processed

Depth-first search starting from vertex g.

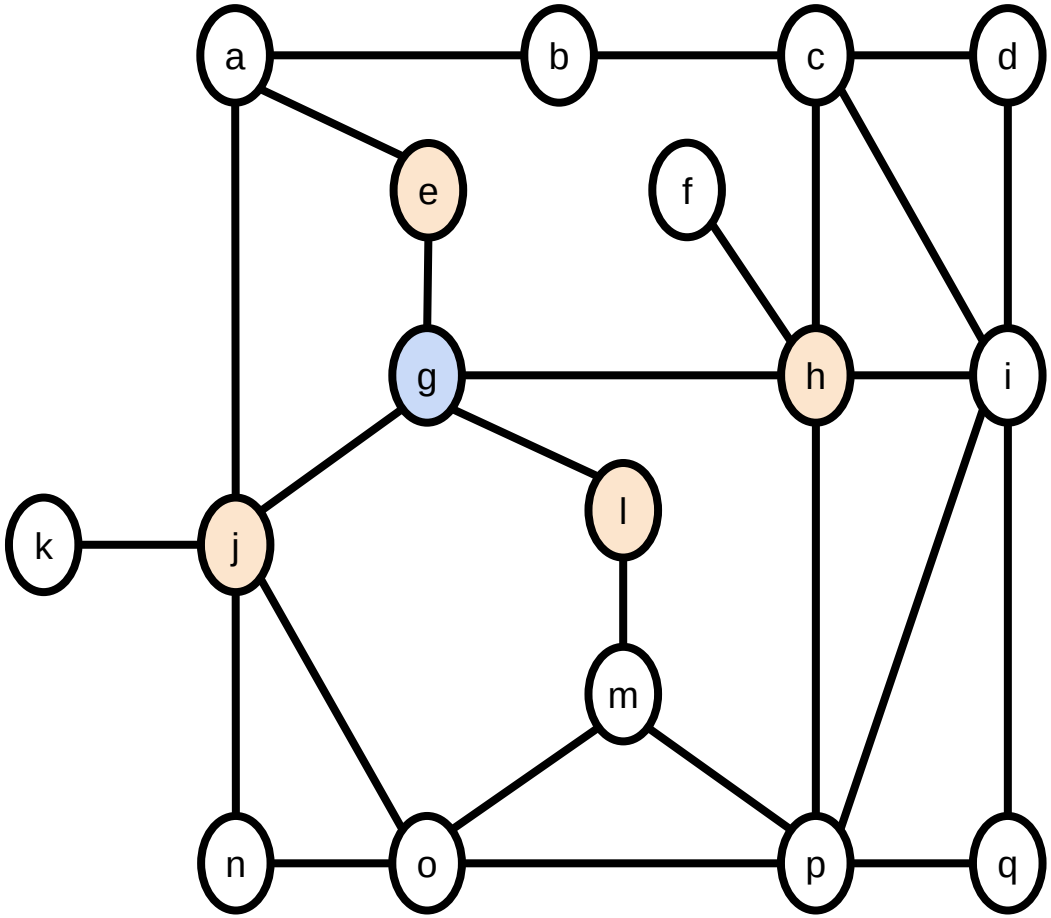


current_vertex | g | todo | e | h | l | j | | | | |

Example: DFS

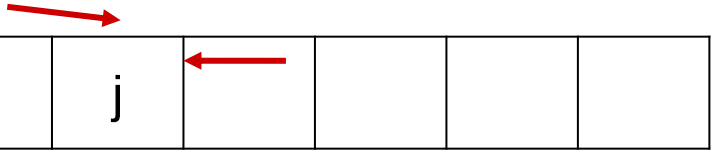
Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



current_vertex

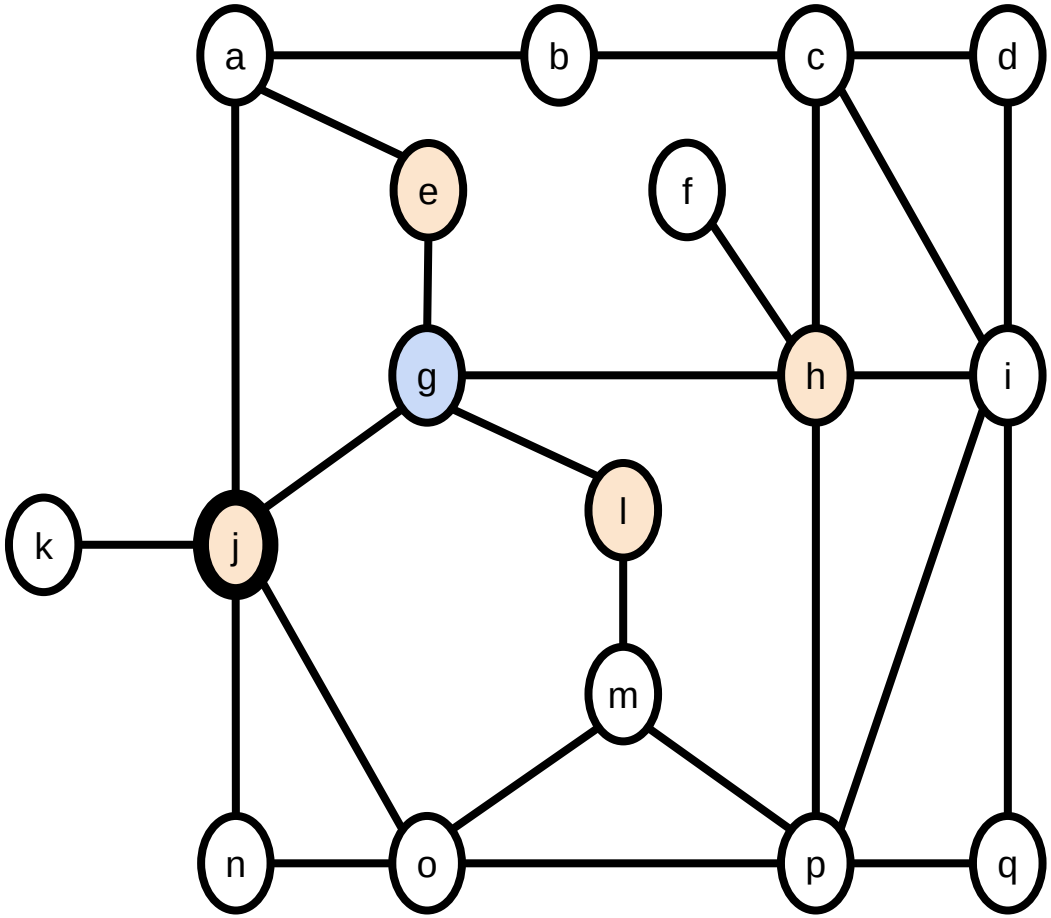
todo



Example: DFS

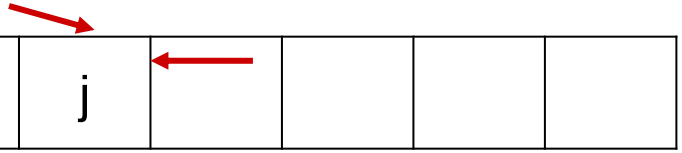
Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



current_vertex

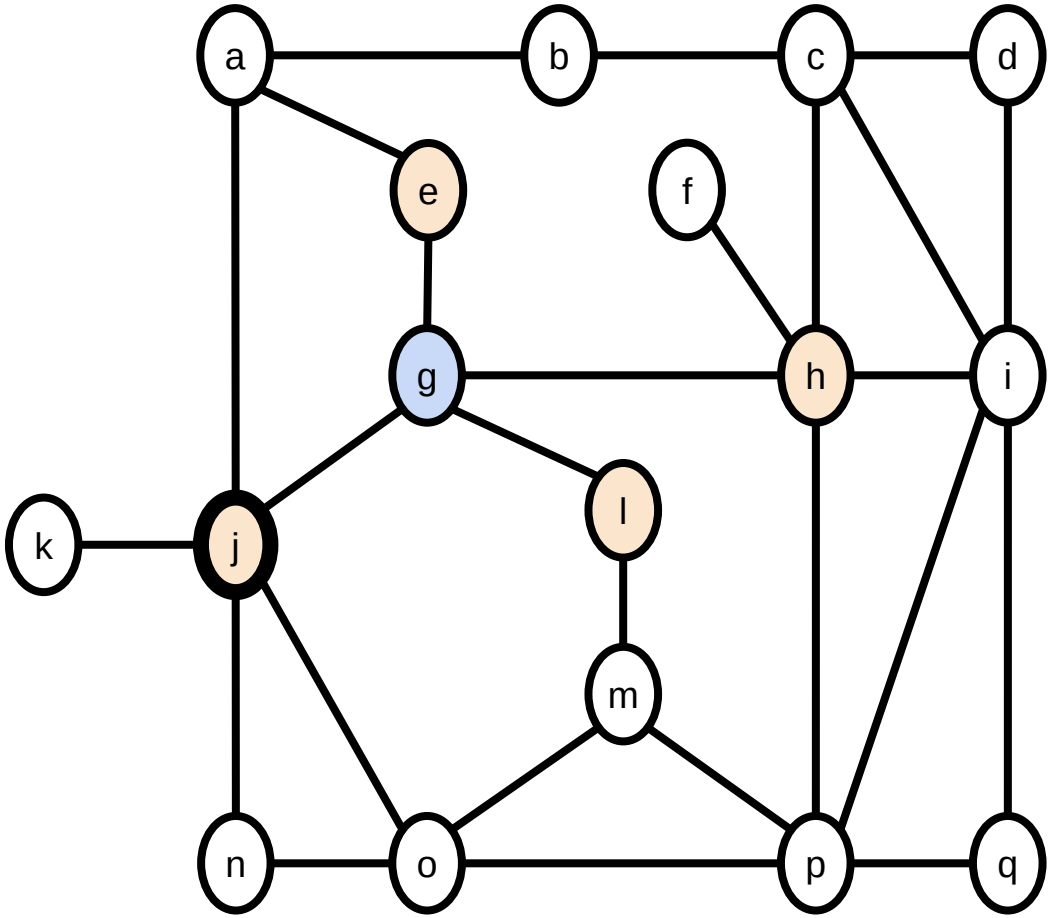
todo



Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

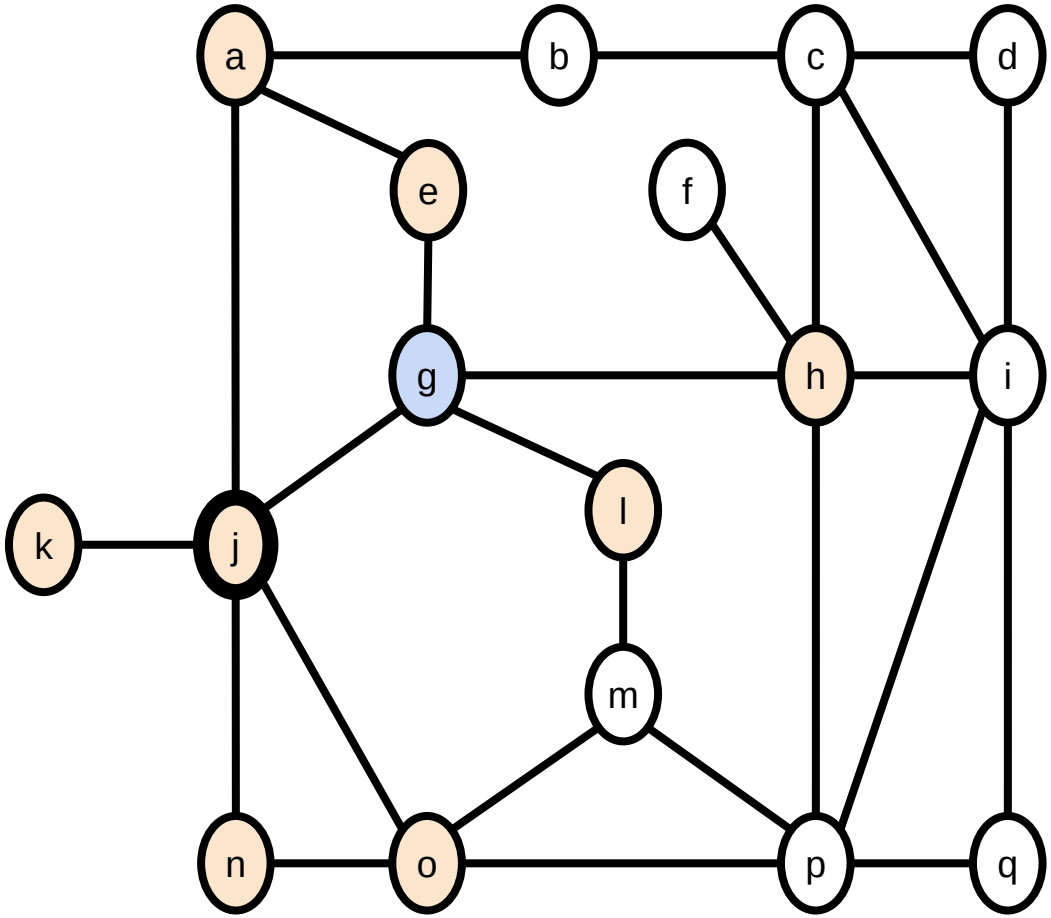


current_vertex | j | todo | e | h | l | | | | | |

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

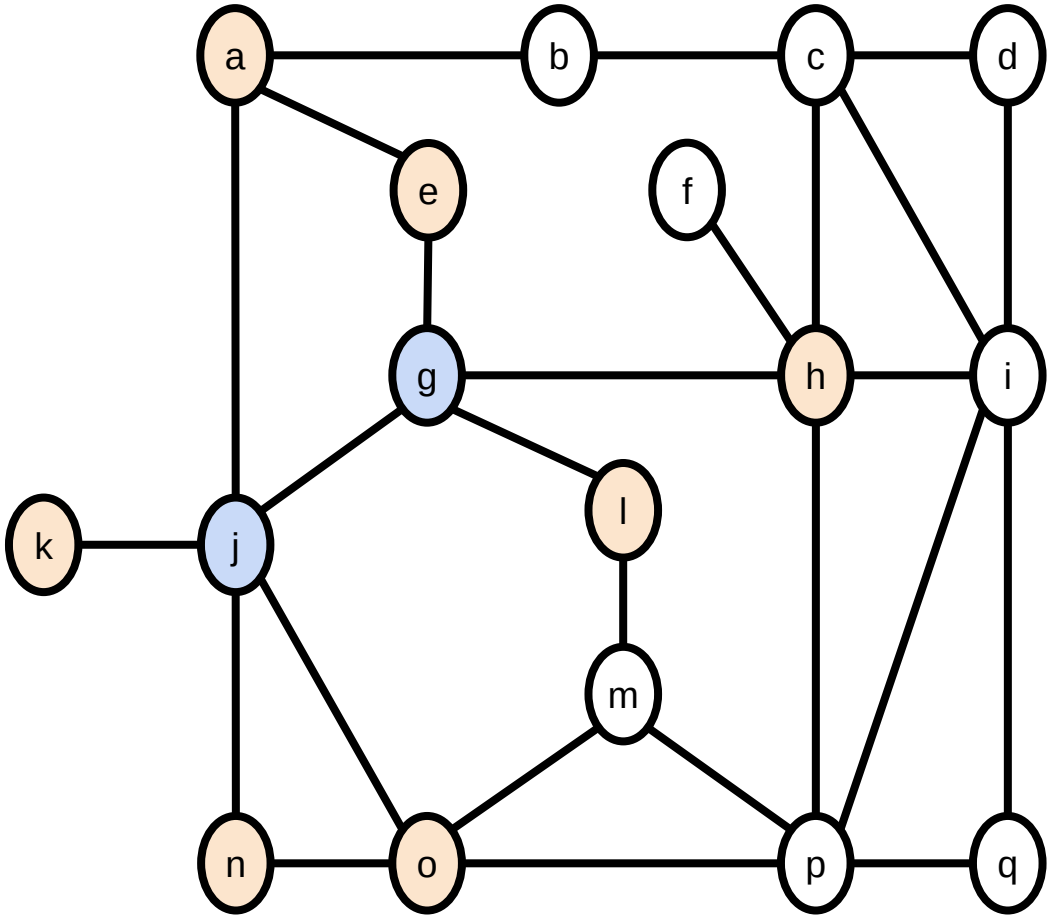


current_vertex | j | todo | e | h | l | a | o | n | k |

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

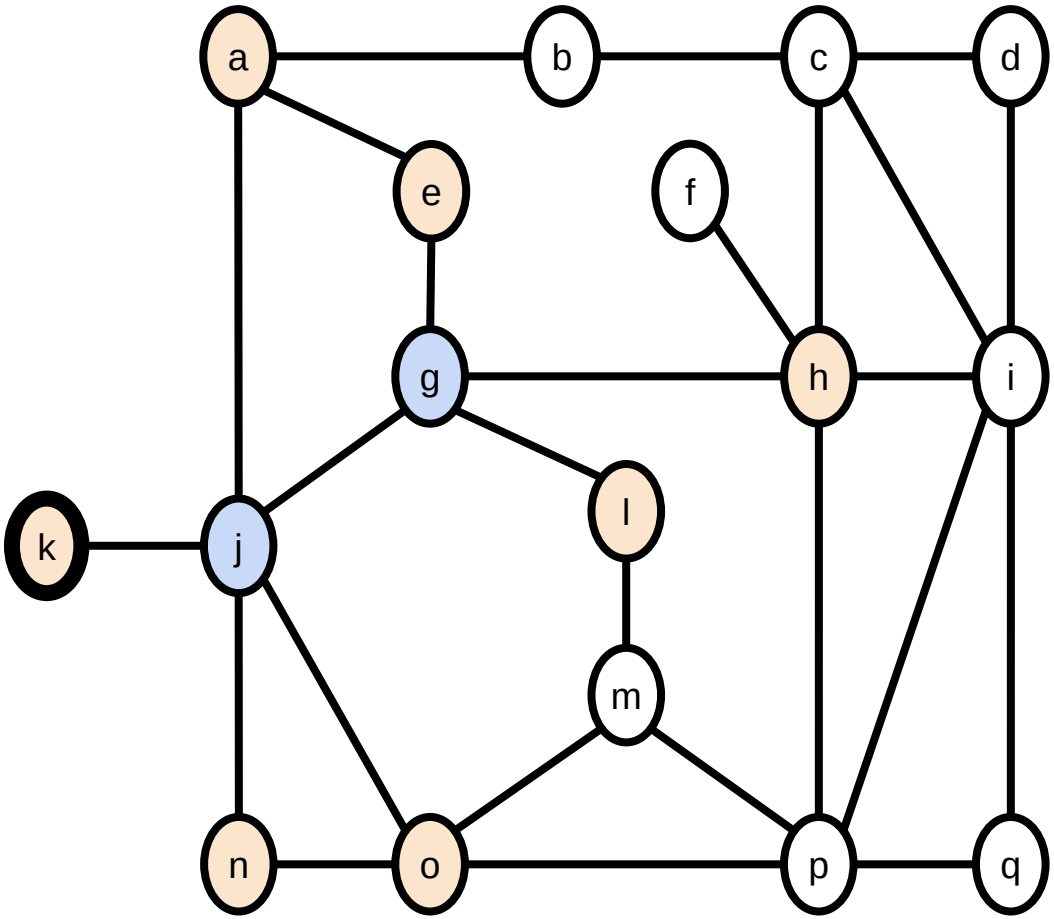


current_vertex [ ] todo [e | h | l | a | o | n | k]

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



current_vertex

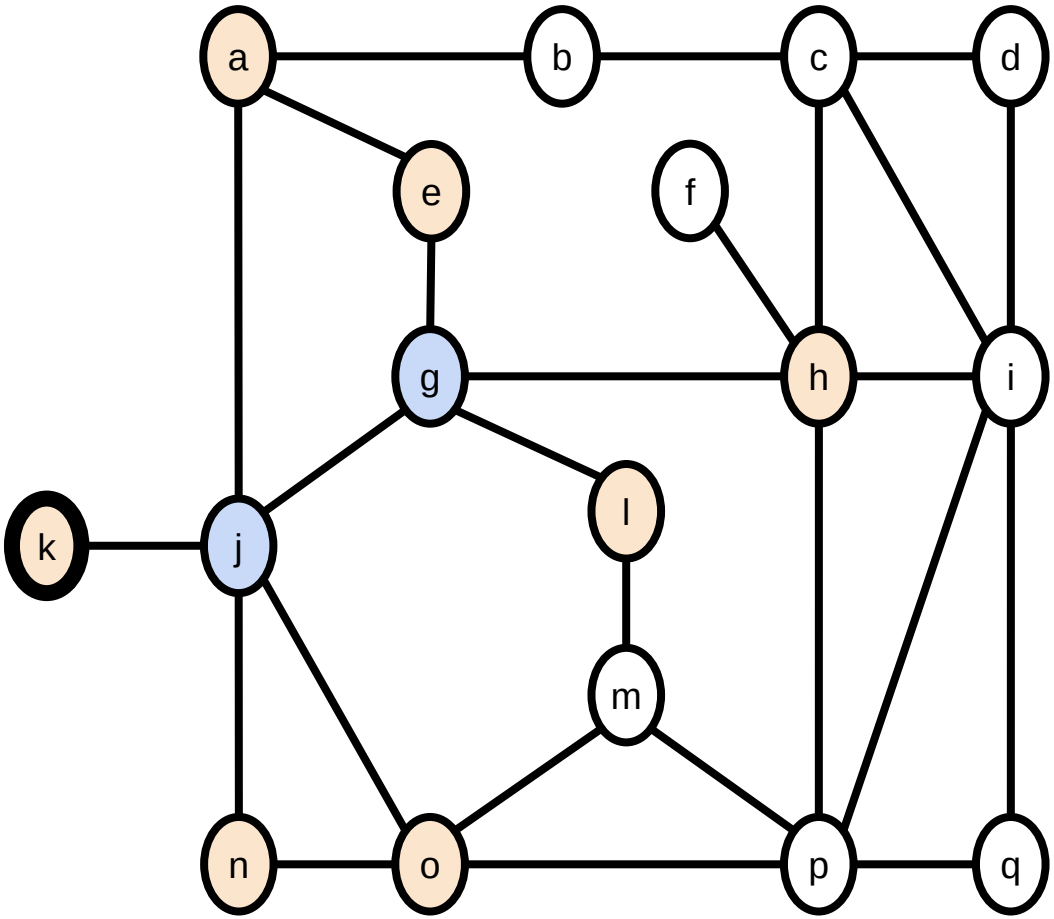
todo

e h l a o n k

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

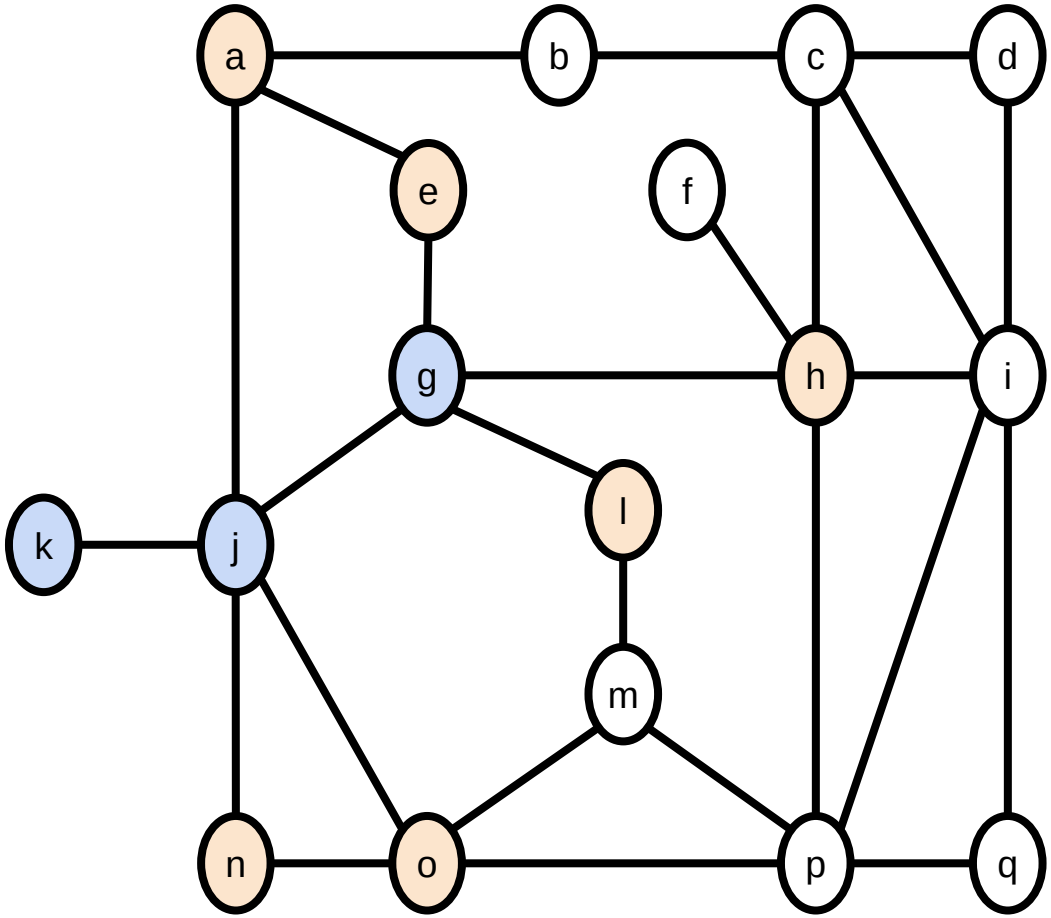


current_vertex | k | todo | e | h | l | a | o | n |

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

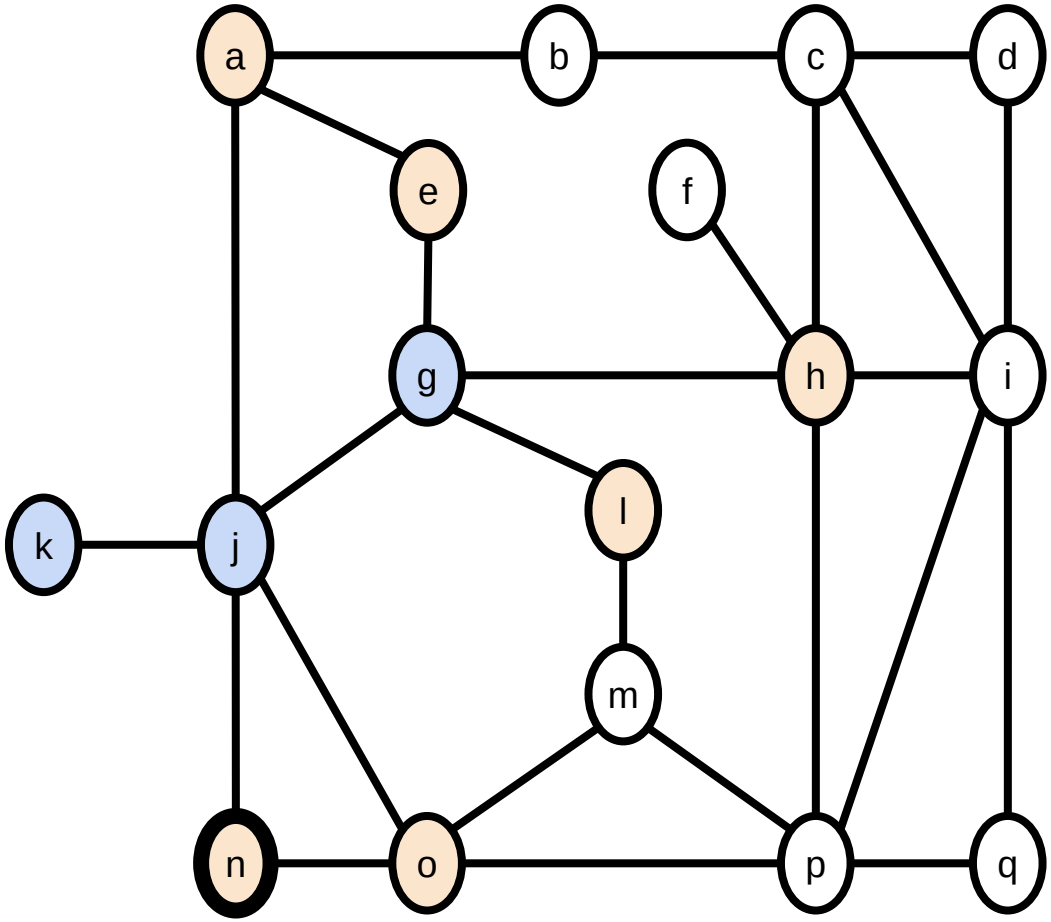


current_vertex [ ] todo [e | h | l | a | o | n | ← |]

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

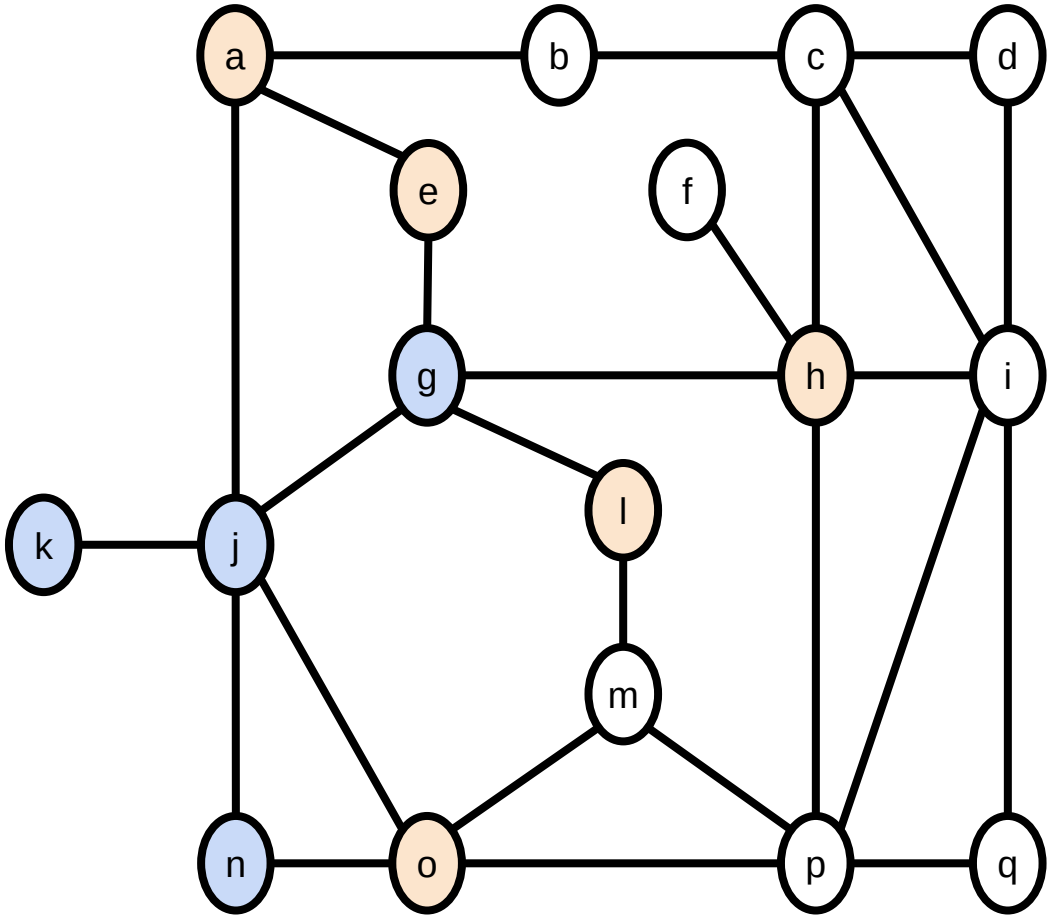


current_vertex | n | todo | e | h | l | a | o | | | |

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

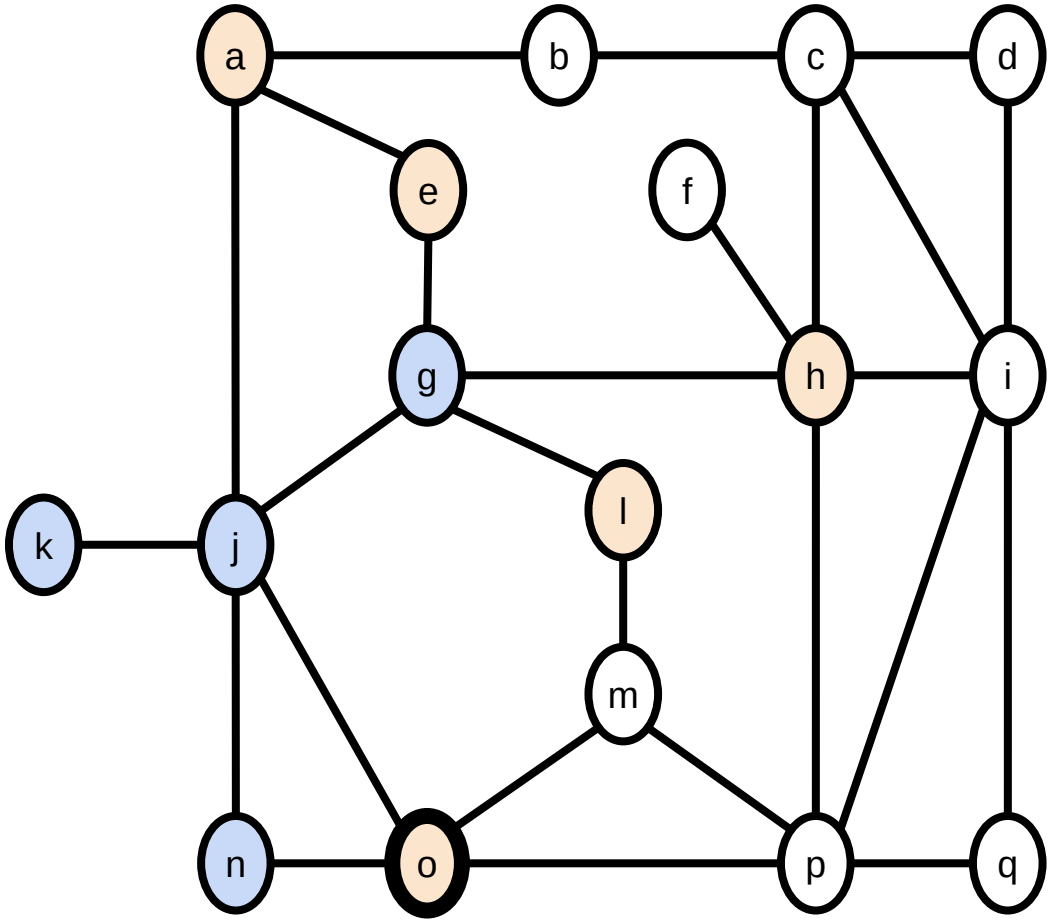


current_vertex | | todo | e | h | l | a | o | | | |

Example: DFS

color	state
	undiscovered
	discovered
	processed

Depth-first search starting from vertex g.

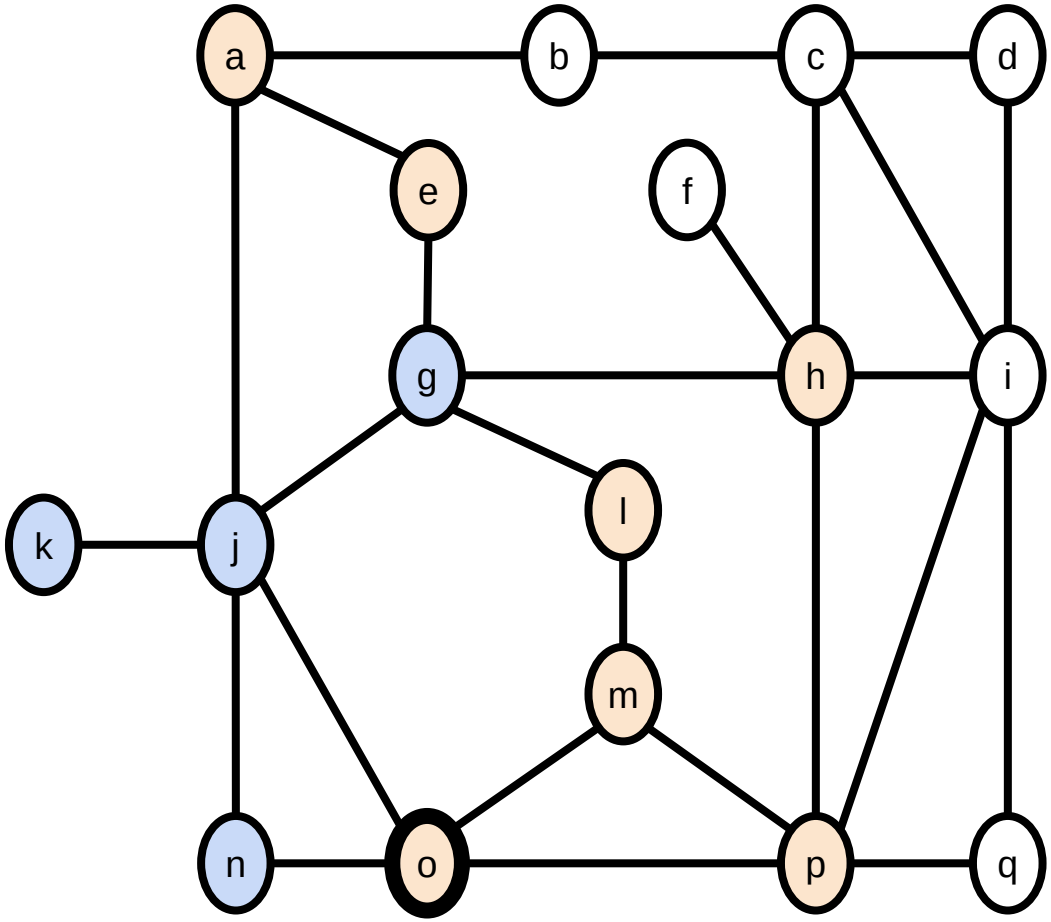


current_vertex | o | todo | e | h | l | a |

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

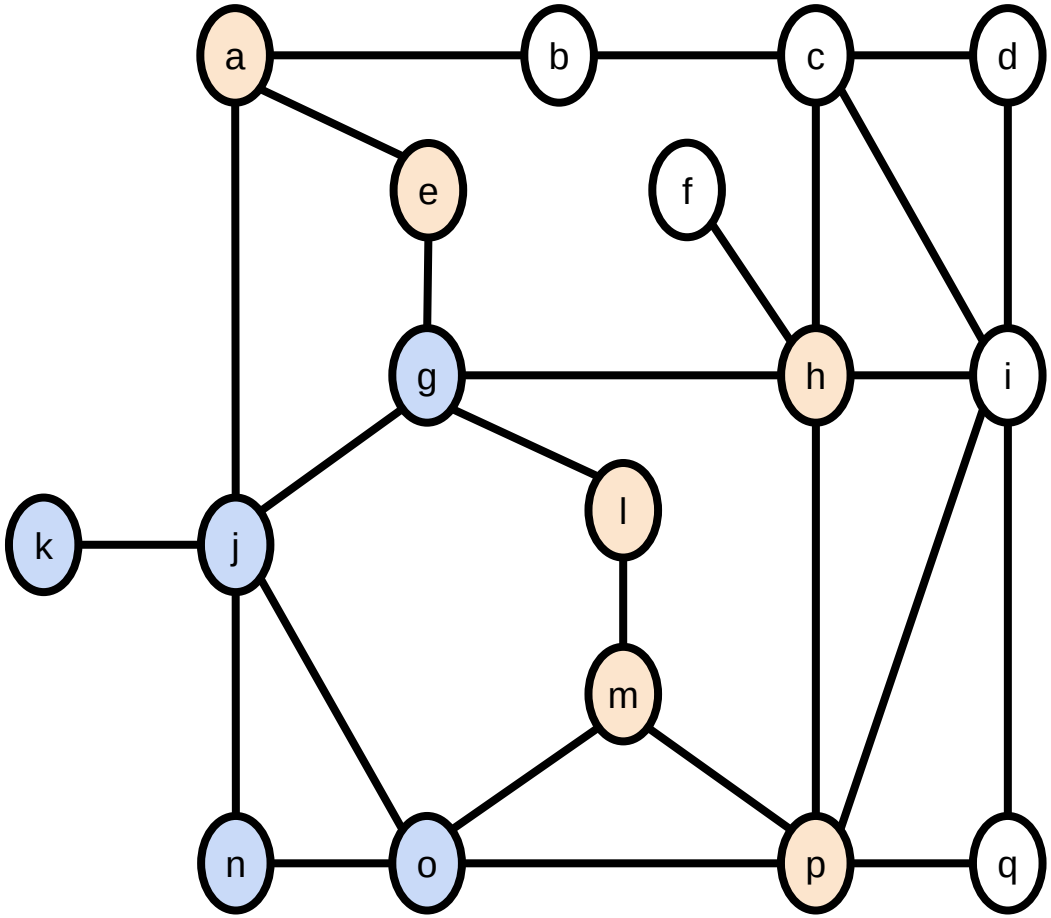


current_vertex | o | todo | e | h | l | a | m | p | ← |

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

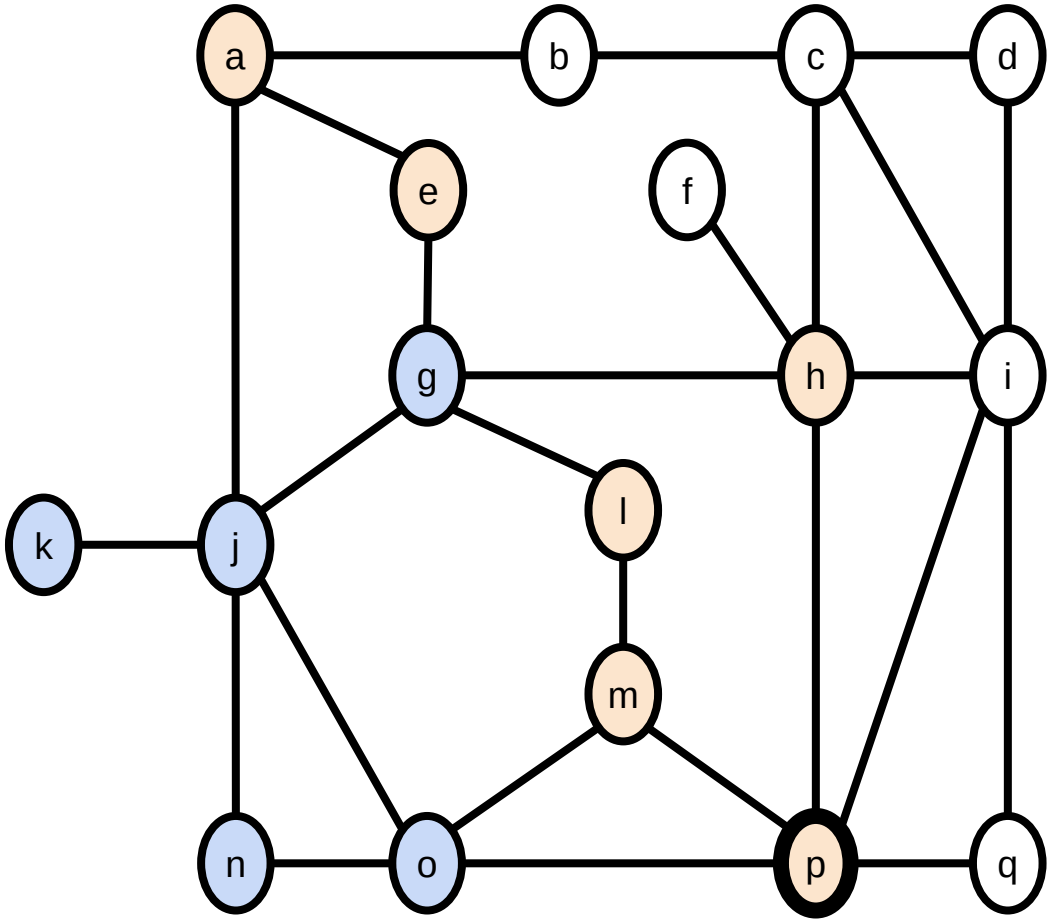


current_vertex | | todo | e | h | l | a | m | p | ← |

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

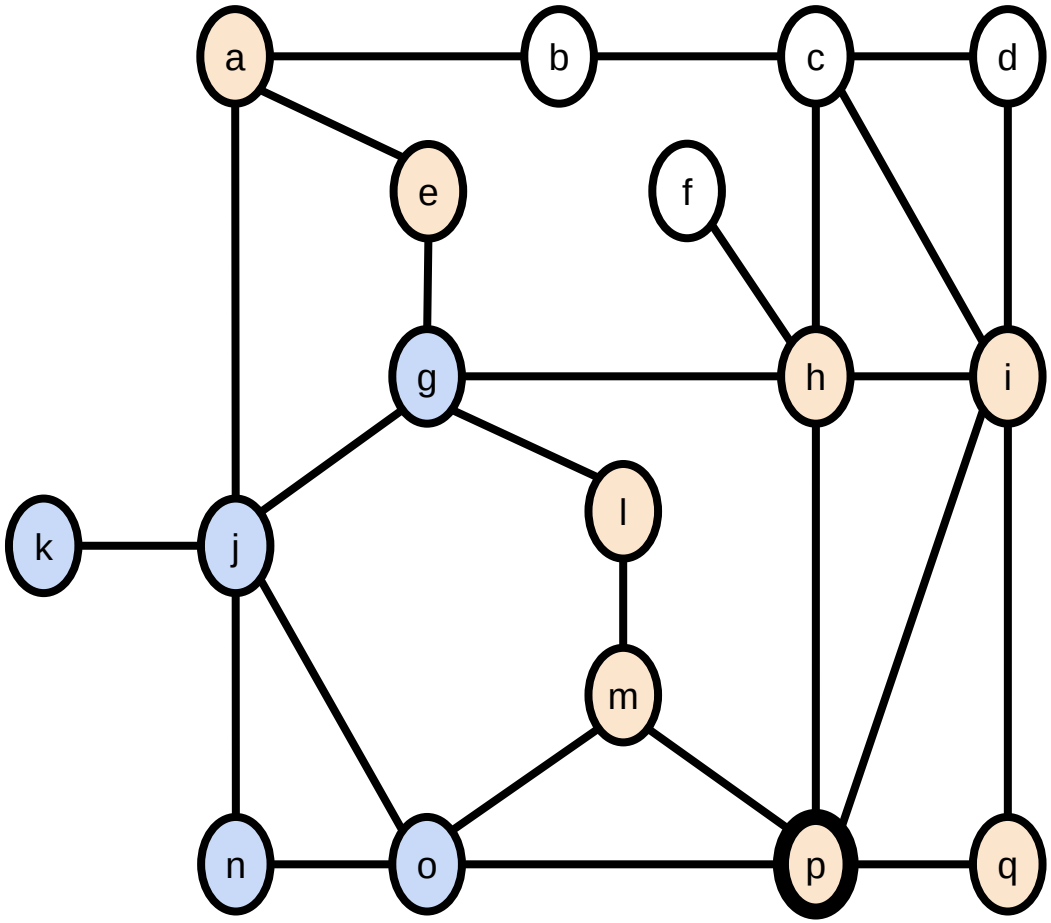


current_vertex | p | todo | e | h | l | a | m | | | |

Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed

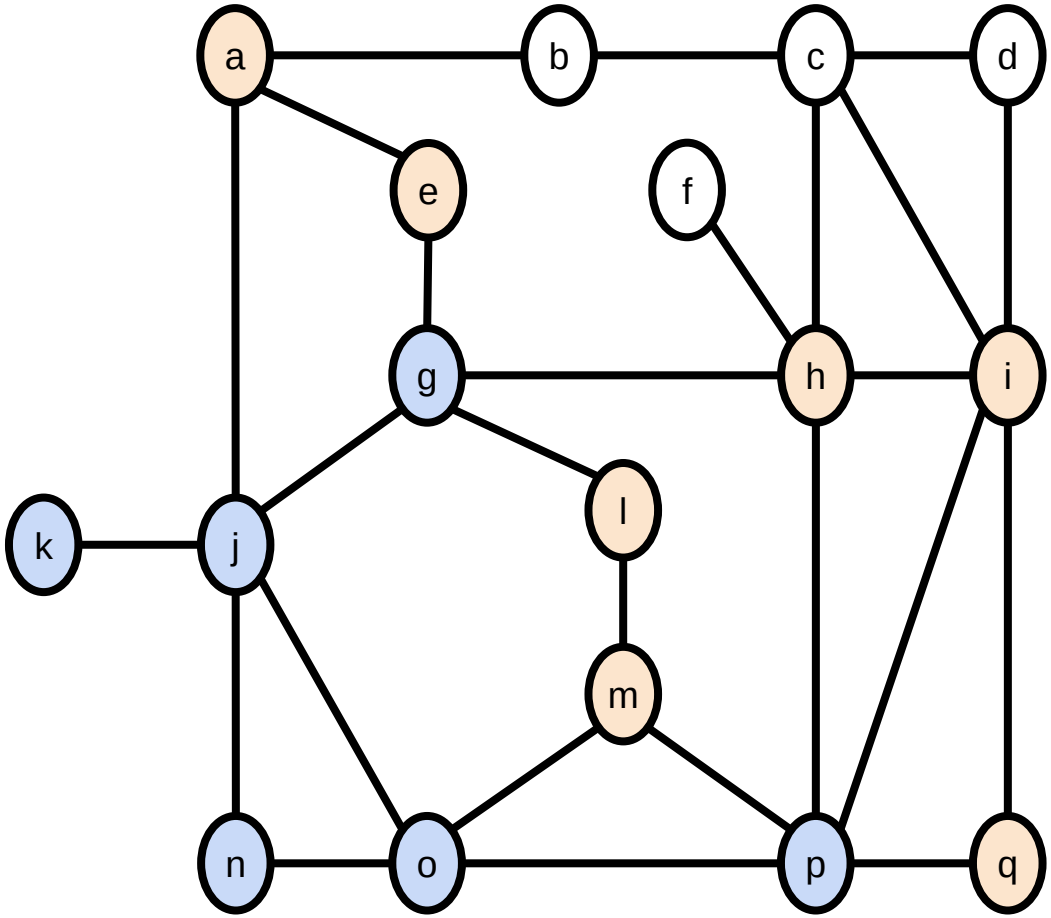


current_vertex | p | todo | e | h | l | a | m | i | q |

Example: DFS

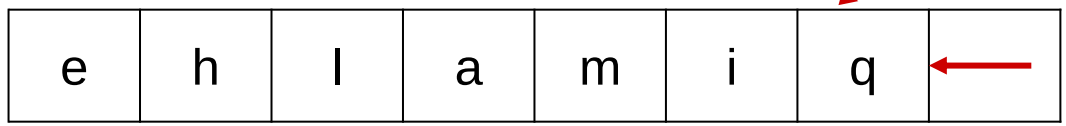
Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



current_vertex

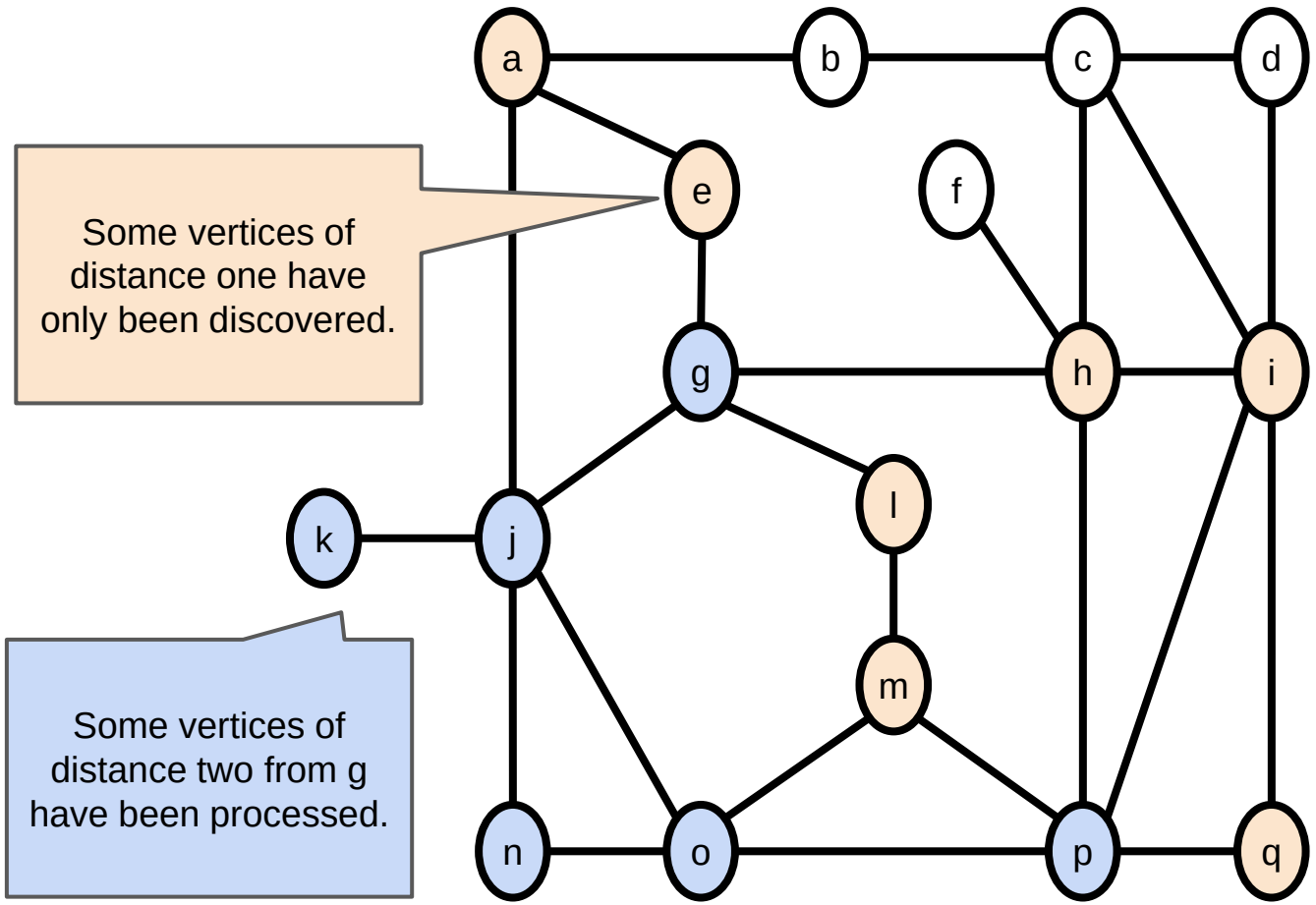
todo



Example: DFS

Depth-first search starting from vertex g.

color	state
	undiscovered
	discovered
	processed



current_vertex | | todo | e | h | l | a | m | i | q |

DFS Notes

Note: on the exam you could be asked for the order that the nodes are discovered / processed. In the previous example the answer would be the following.

g	e	h	l	j	a	o	n	k	m	p	i	q	d	c	b	f

discovered

g	j	k	n	o	p	q	i	c	b	d	m	a	l	h	f	e

processed

Breadcrumb (DFS) algorithm for getting out of the woods

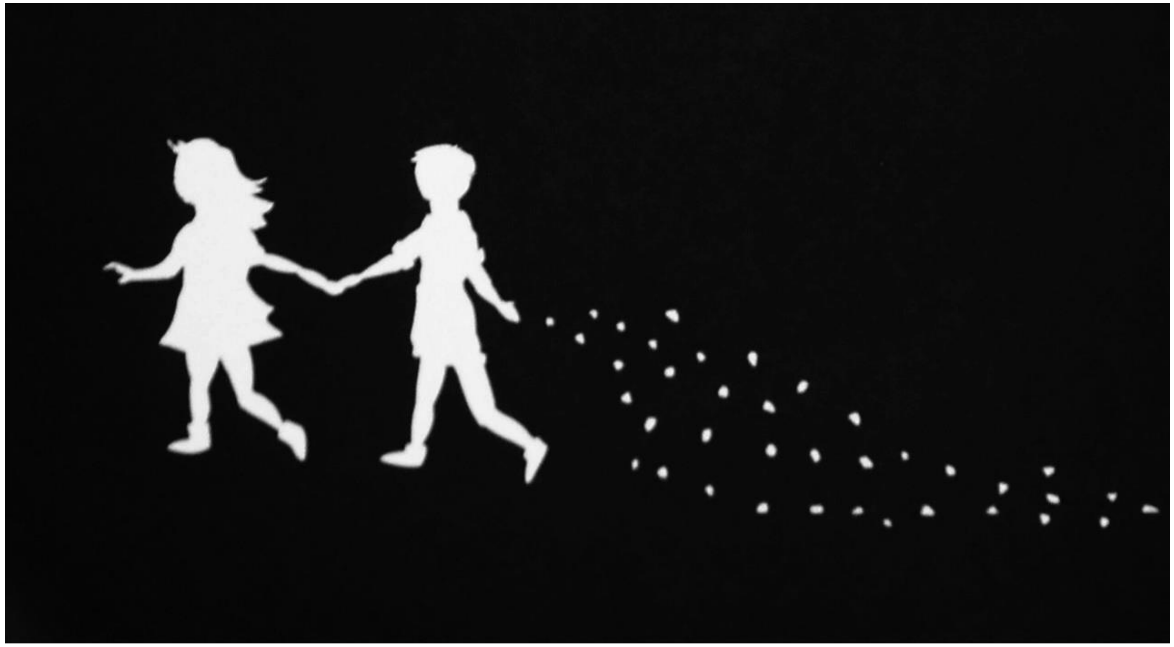
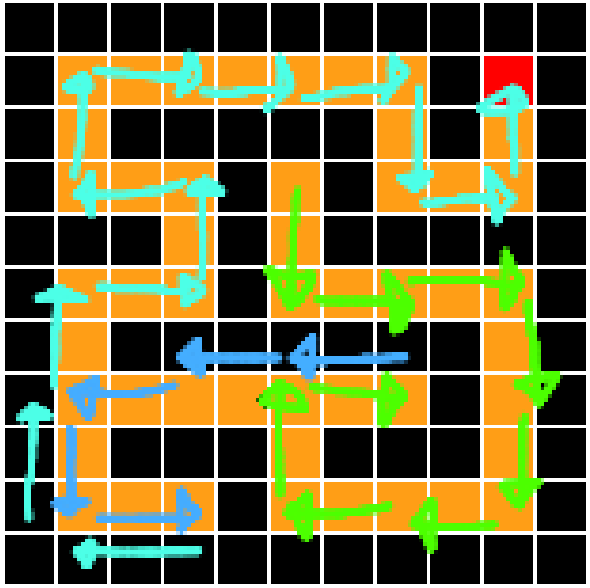


Image of BFS is from wikipedia

Solving mazes with graph traversals



- DFS (as well as BFS) can be used to solve mazes.
- Both algorithms traverse the entire graph - so eventually they will find a path from start to exit, if such path exists.

Depth-First Search (pop only when there is nothing to add)

We can modify the **DFS** to mark the vertex as processed when all vertices reachable from it have been discovered/processed.

Algorithm DFS (graph G, start)

```
1    for each u in vertices of G:
2        u.state:= "undiscovered"

3    start.state:= "discovered"
4    todo:= new stack()
5    todo.push(start)

6    while todo is not empty:
7        current_vertex:= todo.top()    # don't pop yet: just read
8        for each u in neighbors(current_vertex):
9              count: = 0
10             if u.state = "undiscovered":
11                    u.state:= "discovered"
12                    todo.push(u)
13                    count: = count + 1
14       if count = 0:                  # nowhere to go
15             current_vertex.state = "processed"
16             todo.pop()
```

Depth-First Search (Recursive)

This later variation of the Depth-first search can also be implemented recursively.

This implementation implicitly replaces the `todo` stack with the call stack.

```
Algorithm DFS(G, current)
```

```
    current.state:= "discovered"
    for each u in neighbors(current)
        if u.state = "undiscovered" then
            DFS(G, u)
    current.state:="processed"
```

```
for each u in vertices of G
    u.state:= "undiscovered"
DFS(G, start)  // start is a vertex in G
```

Recursive pseudocode for DFS

BFS and DFS: side-by-side

```
ALgorithm BFS(G, start)
 for each u in vertices of G
  u.state:= "un"

 start.state:= "d"

 todo:= new queue()
 todo.enqueue(start)

 while todo is not empty:
  v:= todo.dequeue()
   for each u in neighbors(v)
    if u.state = "un":
     u.state:= "d"
     todo.enqueue(u)

   v.state:= "p"
```

```
ALgorithm DFS(G, start)
 for each u in vertices of G
  u.state:= "un"

 start.state:= "d"

 todo:= new stack()
 todo.push(start)

 while todo is not empty:
  v:= todo.pop()
   for each u in neighbors(v)
    if u.state = "un":
     u.state:= "d"
     todo.push(u)

   v.state:= "p"
```

Runtime of BFS and DFS

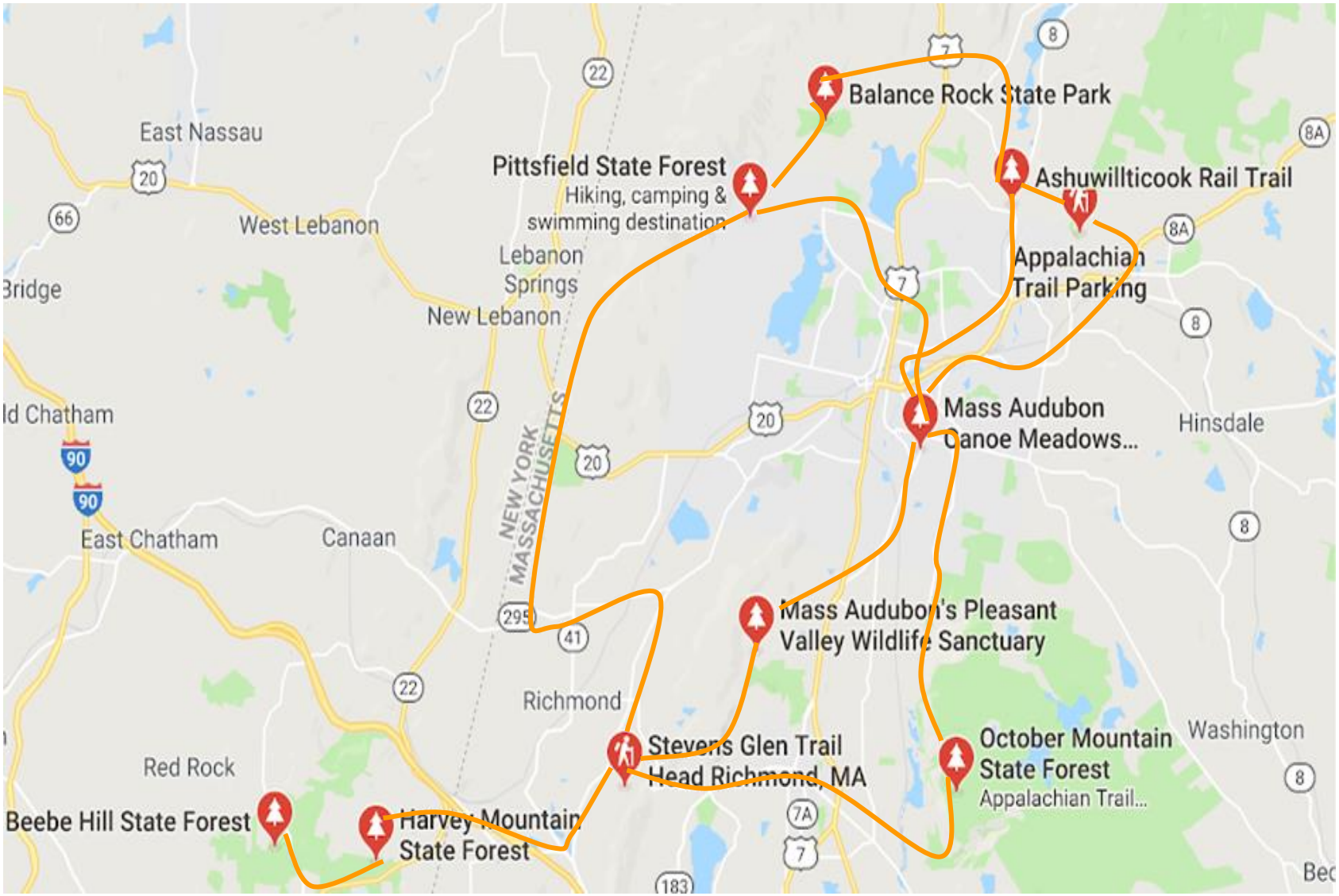
Both algorithms are O(n+m)-time, where m is the number of edges. There are at most $O(n^2)$ edges, so this is $O(n^2)$-time.

```
ALgorithm BFS(G, start)
 for each u in vertices of G
  u.state:= "un"

 start.state:= "d"

 todo:= new queue()
 todo.enqueue(start)

 while todo is not empty:
  v:= todo.dequeue()
   for each u in neighbors(v)
    if u.state = "un":
     u.state:= "d"
     todo.enqueue(u)

   v.state:= "p"
```

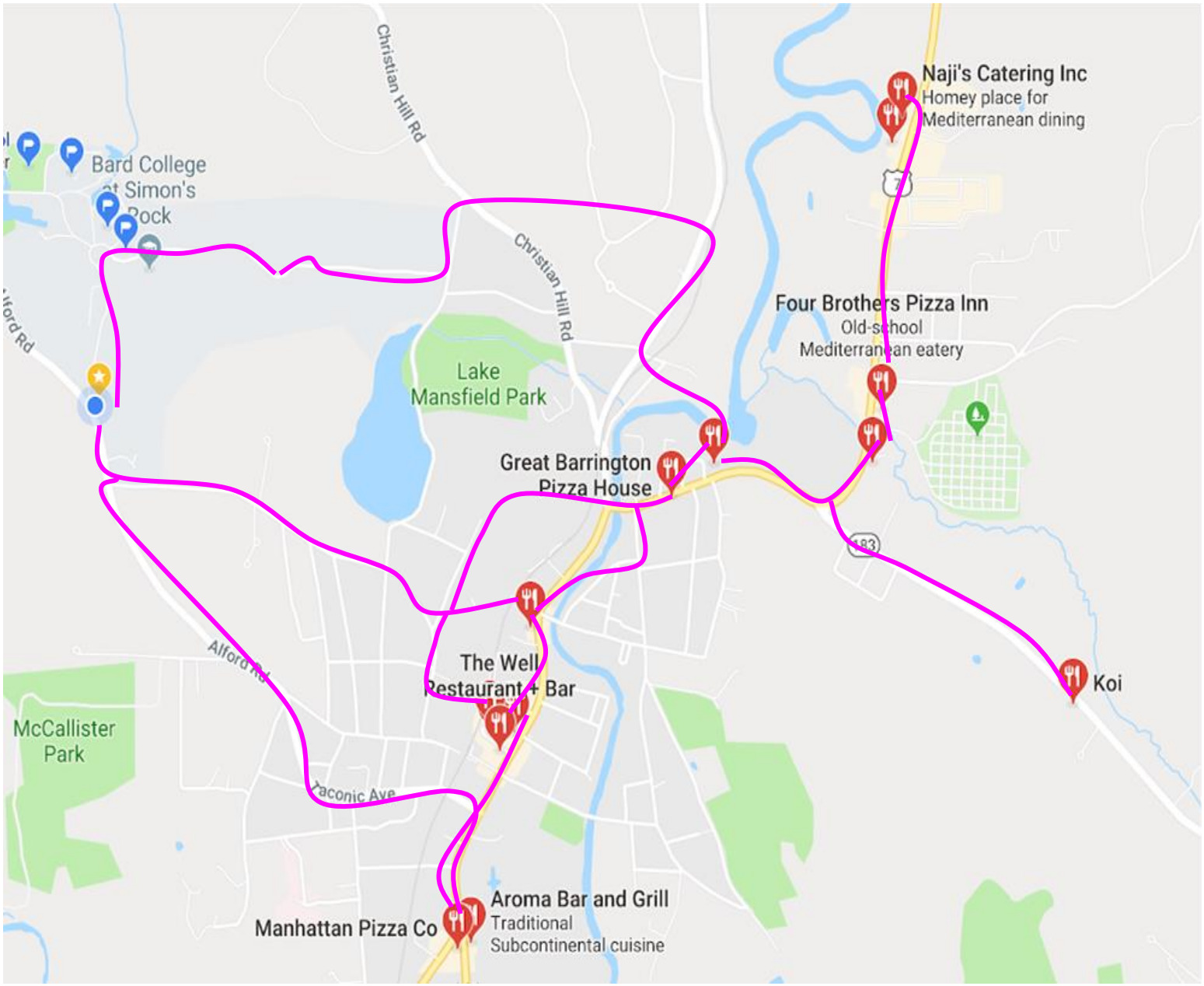
```
ALgorithm DFS(G, start)
 for each u in vertices of G
  u.state:= "un"

 start.state:= "d"

 todo:= new stack()
 todo.push(start)

 while todo is not empty:
  v:= todo.pop()
   for each u in neighbors(v)
    if u.state = "un":
     u.state:= "d"
     todo.push(u)

   v.state:= "p"
```

Notice that the stack and queue operations must guarantee O(1)-time.

Exploration strategies: hiking trails



Exploration strategies: food



Exploration strategies

Some people fully explore areas around them before moving on, and others continue moving further and further from their home, and return to explore the surrounding areas later.

These strategies roughly correspond to ***breadth-first*** and ***depth-first*** respectively.