CSC 225 - Summer 2019

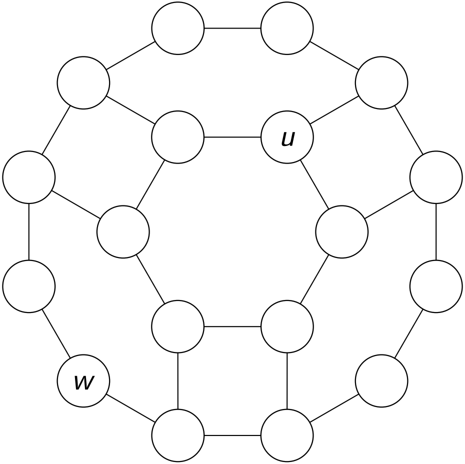
Traversals I

Bill Bird

Department of Computer Science
University of Victoria

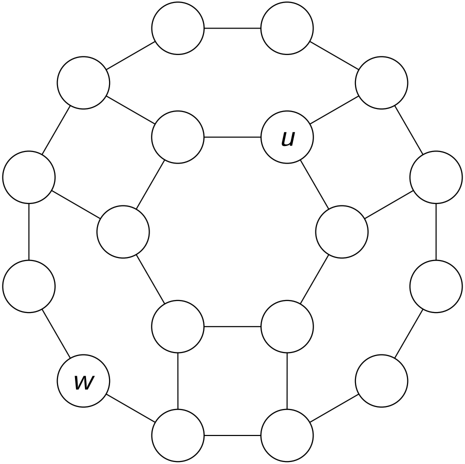
July 17, 2019

Path Finding (1)



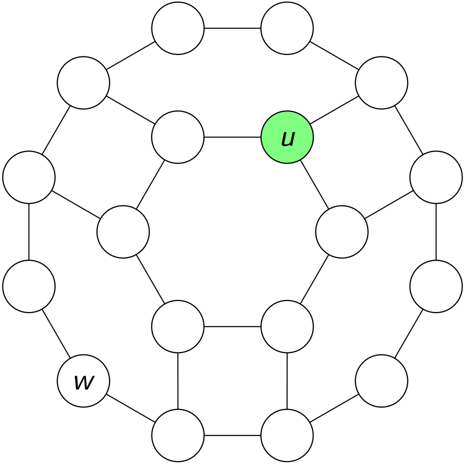
Problem: Design an algorithm to find a path from u to w in the graph above.

Path Finding (2)



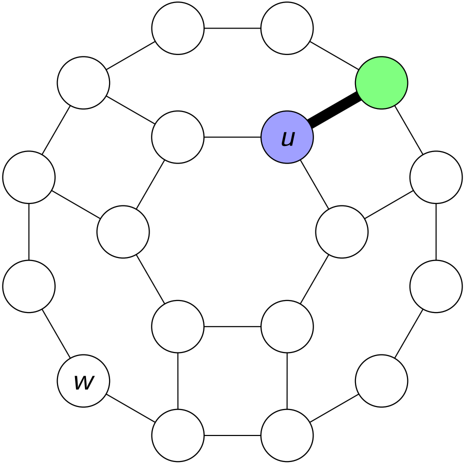
Although it is clear to a human observer how to reach w from u, there is no information in the graph itself besides the neighbours of each vertex.

Path Finding (3)



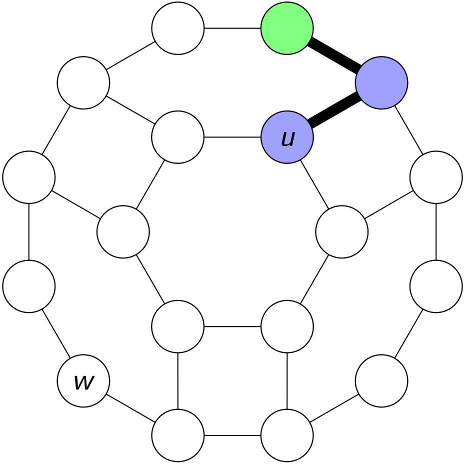
Idea: Start at u, pick a direction and start walking.

Path Finding (4)



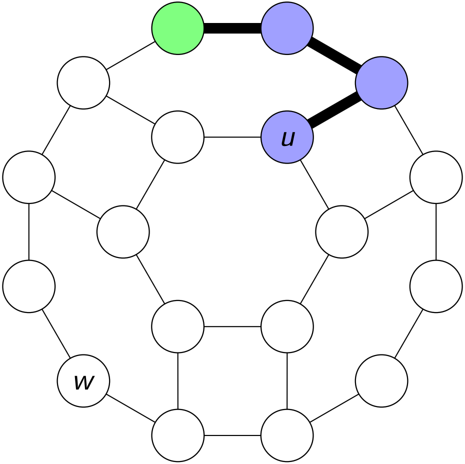
After arriving at a neighbour of u, mark u as 'visited' and keep track of the edge used.

Path Finding (5)



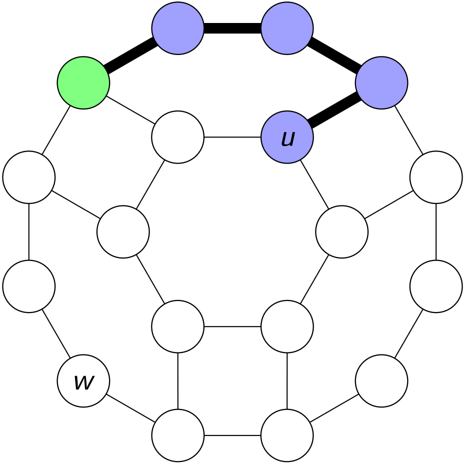
Continue walking through the graph, picking an unvisited vertex (white) at each step, and marking all visited vertices (blue).

Path Finding (6)



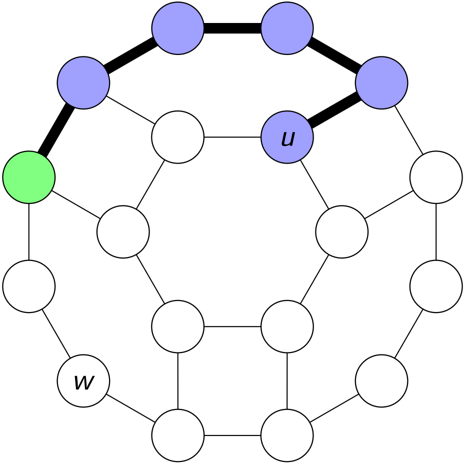
Continue walking through the graph, picking an unvisited vertex (white) at each step, and marking all visited vertices (blue).

Path Finding (7)



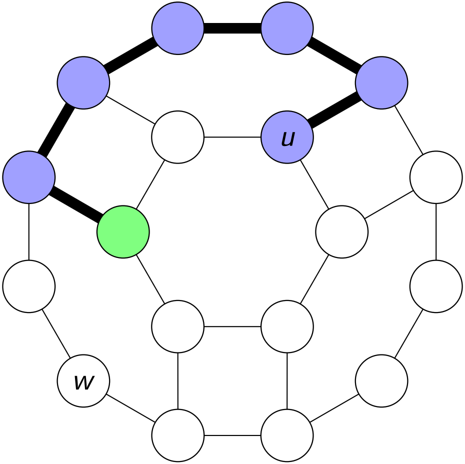
Continue walking through the graph, picking an unvisited vertex (white) at each step, and marking all visited vertices (blue).

Path Finding (8)



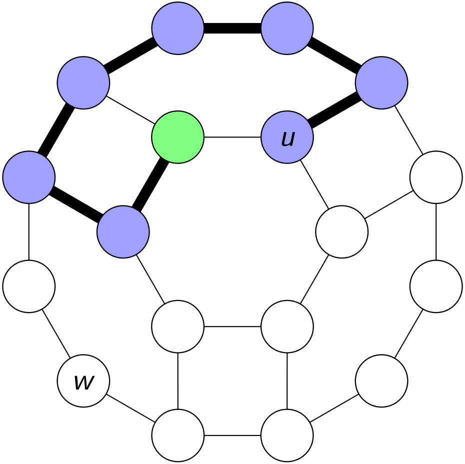
Continue walking through the graph, picking an unvisited vertex (white) at each step, and marking all visited vertices (blue).

Path Finding (9)



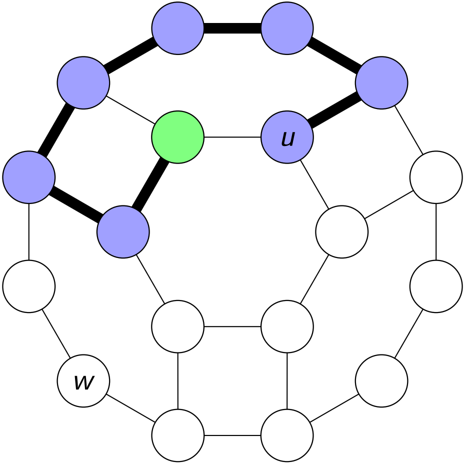
Continue walking through the graph, picking an unvisited vertex (white) at each step, and marking all visited vertices (blue).

Path Finding (10)



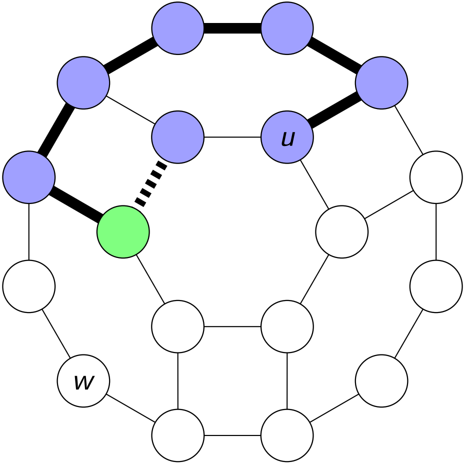
At this point, there are no unvisited vertices neighbouring the current vertex.

Path Finding (11)



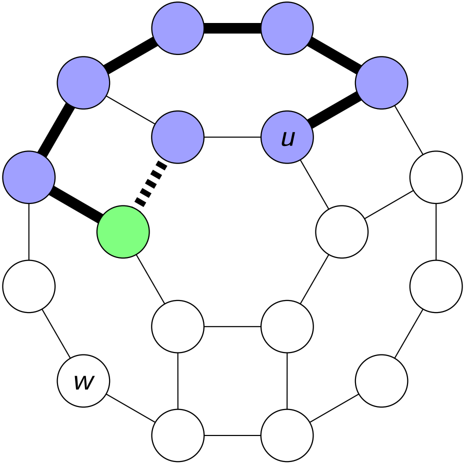
It is impossible to continue without either visiting an already-visited neighbour or doubling back.

Path Finding (12)



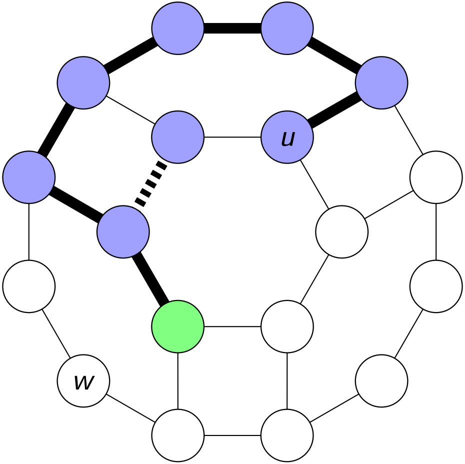
Since the path reached a dead end, double back and ignore the edge used (but leave the vertex marked as visited).

Path Finding (13)

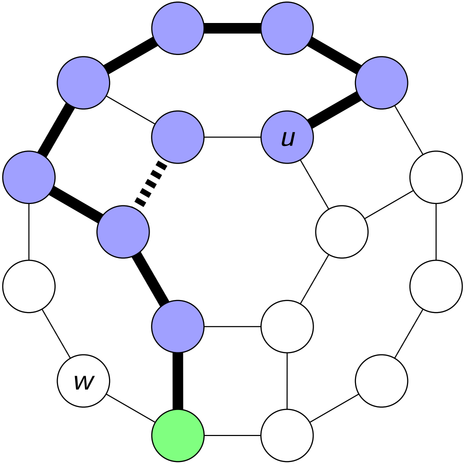


After backing up to the last vertex with an unvisited neighbour, continue walking.

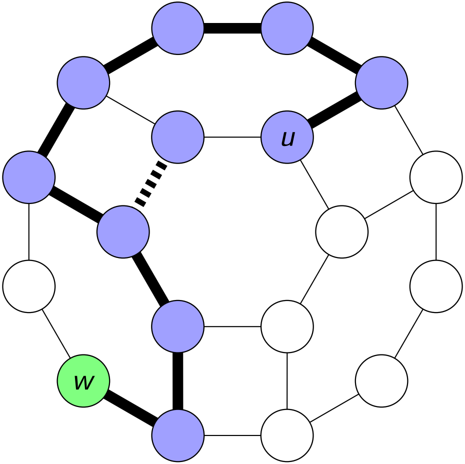
Path Finding (14)



Path Finding (15)

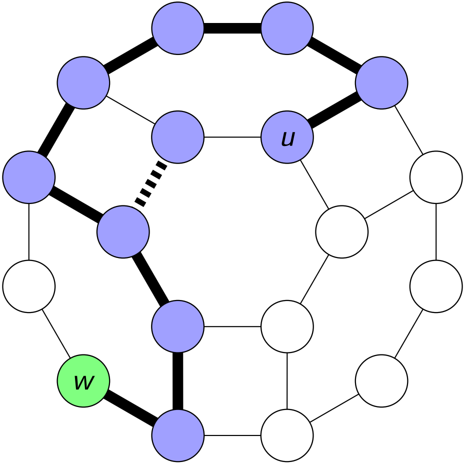


Path Finding (16)



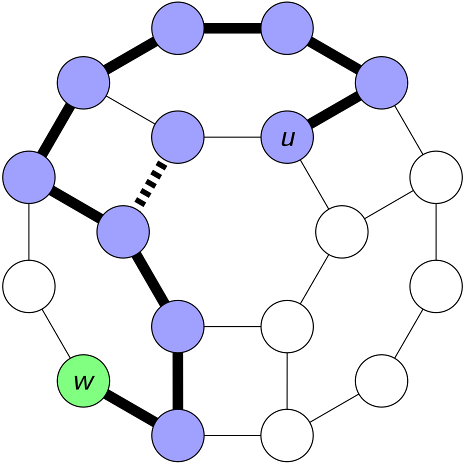
When w is reached, the set of edges followed during the walk (darkened in the diagram) is a path from u to w.

Path Finding (17)



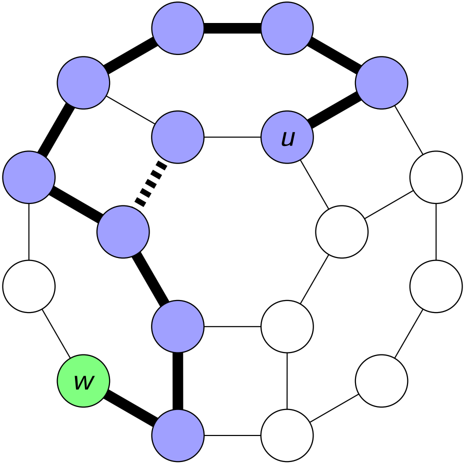
Observe that the resulting path is not the most direct route.

Path Finding (18)



Algorithms which explore a graph by walking along edges are called **graph traversals**.

Path Finding (19)



The 'pick a direction and start walking' approach corresponds to an algorithm called **Depth First Search** (or DFS).

Path Finding (20)

```
procedure PathDFS(v, w, PathStack)
    if v is marked as visited then
        return
    end if
    Mark v as visited
    Push v onto PathStack
    if v = w then
        Output the contents of PathStack
        return
    end if
    for each neighbour q of v do
        PathDFS(q, w, PathStack)
    end for
    Pop PathStack
end procedure
```

The pseudocode above describes the algorithm in the example.

Path Finding (21)

```
procedure PathDFS(v, w, PathStack)
    if v is marked as visited then
        return
    end if
    Mark v as visited
    Push v onto PathStack
    if v = w then
        Output the contents of PathStack
        return
    end if
    for each neighbour q of v do
        PathDFS(q, w, PathStack)
    end for
    Pop PathStack
end procedure
```

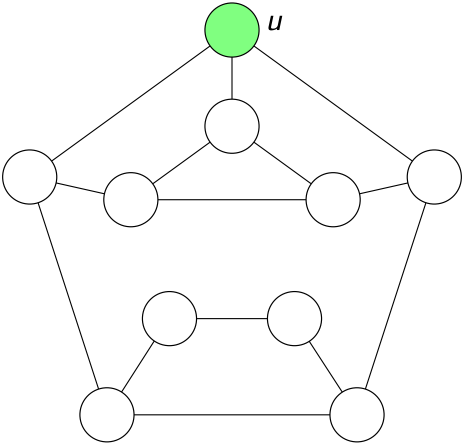
To find a u-w path, the initial call is $\textsc{PathDFS}(u, w, \texttt{PathStack})$, where `PathStack` is an empty stack.

Path Finding (22)

```
procedure PathDFS(v, w, PathStack)
    if v is marked as visited then
        return
    end if
    Mark v as visited
    Push v onto PathStack
    if v = w then
        Output the contents of PathStack
        return
    end if
    for each neighbour q of v do
        PathDFS(q, w, PathStack)
    end for
    Pop PathStack
end procedure
```

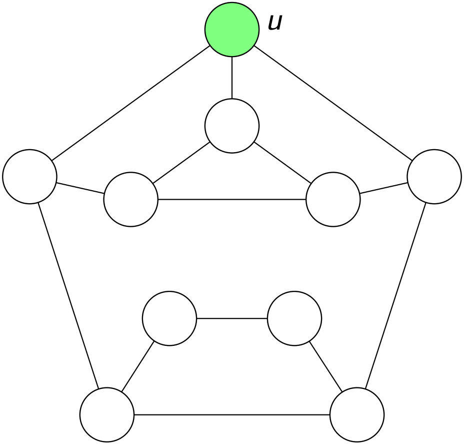
When the traversal reaches vertex w, the contents of `PathStack` will be the set of vertices in a u-w path.

Depth-First Search (1)



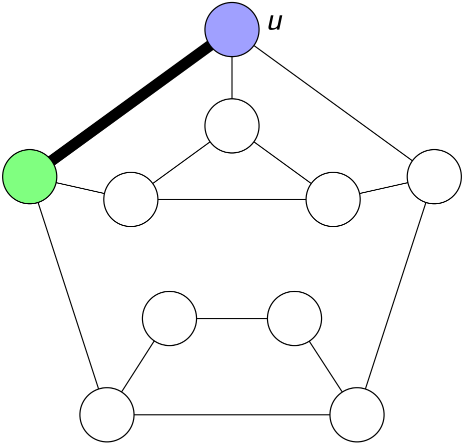
The path finding algorithm in the previous example was a special case of Depth-First Search (DFS).

Depth-First Search (2)



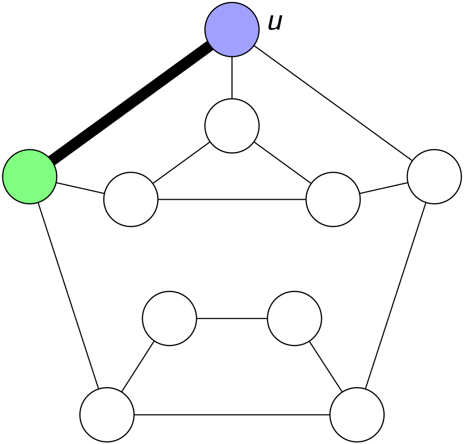
The general DFS algorithm starts at a vertex u and visits every vertex reachable from u.

Depth-First Search (3)



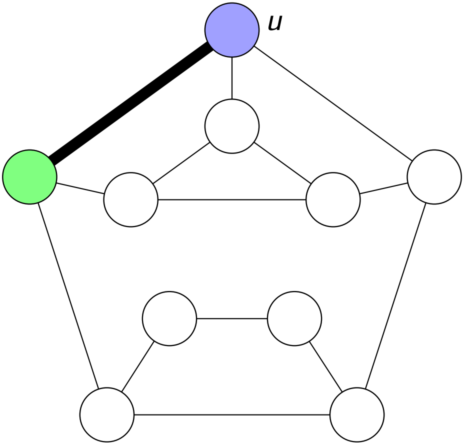
At each step, a neighbour of the current vertex is chosen to be the next vertex visited.

Depth-First Search (4)



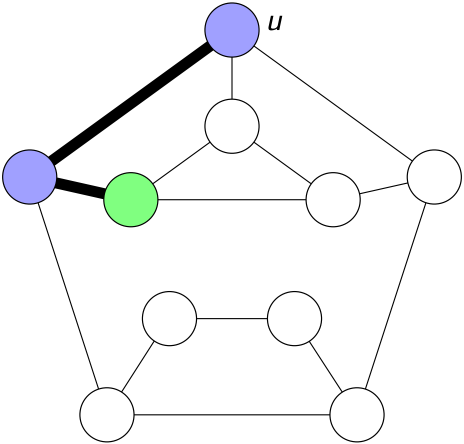
The edge (darkened) followed to reach the next vertex becomes part of the **DFS tree**.

Depth-First Search (5)



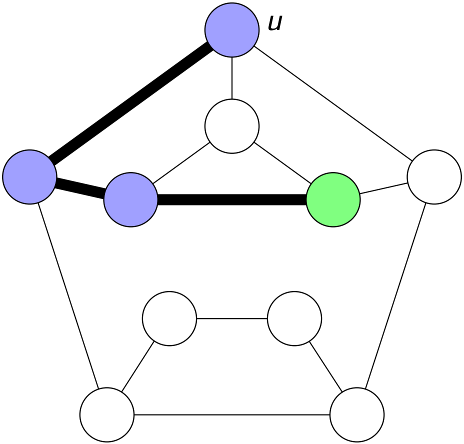
The root of a DFS tree is the initial starting vertex u.

Depth-First Search (6)

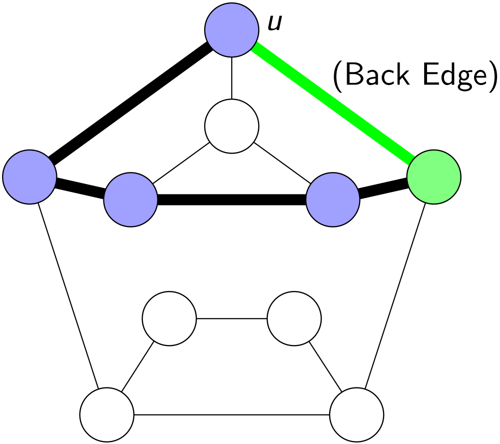


Edges in the DFS tree may be called **tree edges** or **discovery edges**.

Depth-First Search (7)

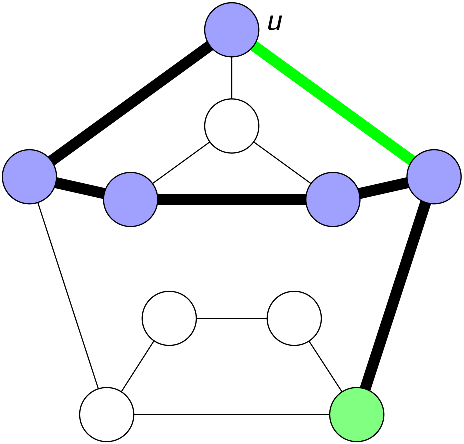


Depth-First Search (8)



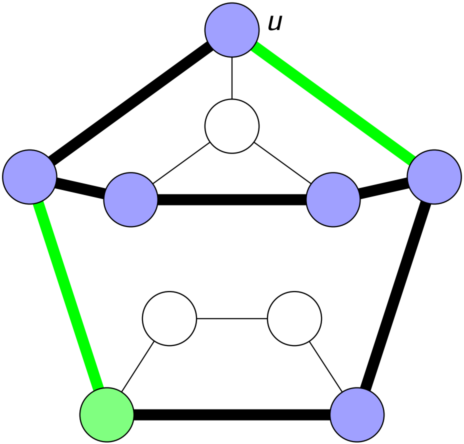
The green edge in the diagram cannot be followed by DFS, since it is connected to a previously visited vertex. Such edges are called **back edges**.

Depth-First Search (9)

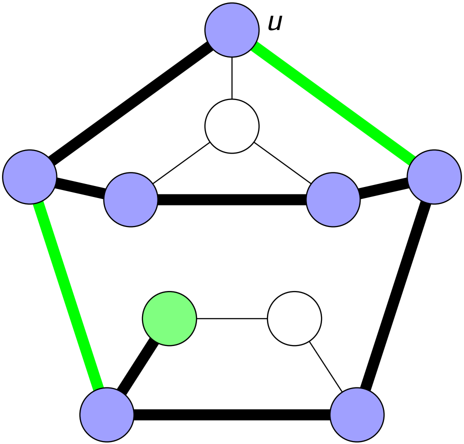


Exercise: Show that if a back edge exists, the graph must contain a cycle.

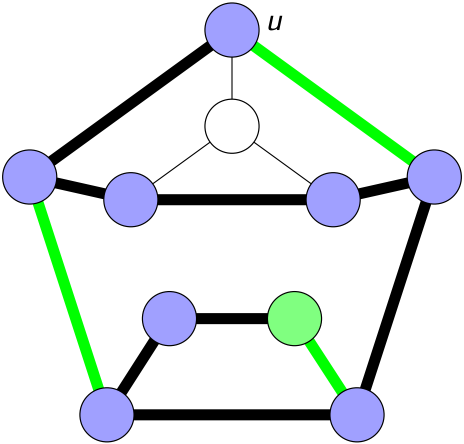
Depth-First Search (10)



Depth-First Search (11)

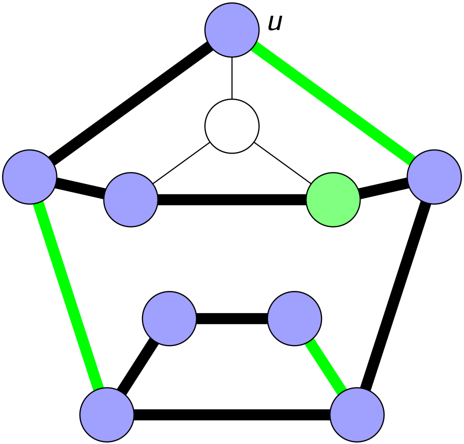


Depth-First Search (12)



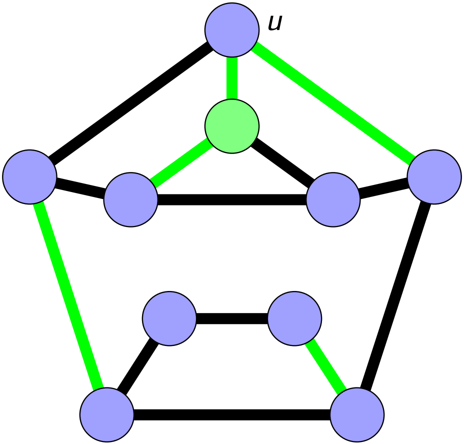
At this point, the current vertex has no unvisited neighbours.

Depth-First Search (13)



Recursion unwinds to the last vertex with unvisited neighbours and the traversal continues from there.

Depth-First Search (14)



A DFS tree is a spanning tree of the connected component containing the starting vertex u.

Depth-First Search (15)

```
procedure DFS(v)
    Mark v as visited
    //Visit v (pre-order)
    for each neighbour w of v do
        if w has not already been visited then
            //(v, w) is a discovery edge
            //Recursively traverse w
            DFS(w)
        else
            //(v, w) is a back edge
        end if
    end for
    //Visit v (post-order)
end procedure
```

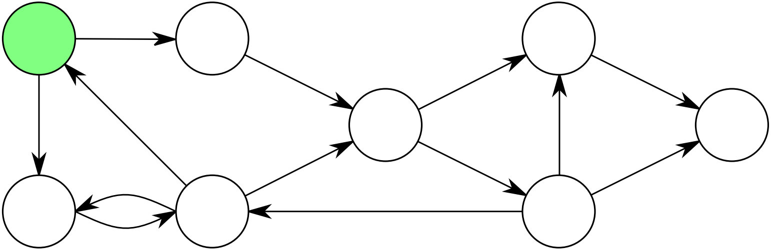
The pseudocode above gives the general structure of DFS.

Depth-First Search (16)

```
procedure DFS(v)
    Mark v as visited
    //Visit v (pre-order)
    for each neighbour w of v do
        if w has not already been visited then
            //(v, w) is a discovery edge
            //Recursively traverse w
            DFS(w)
        else
            //(v, w) is a back edge
        end if
    end for
    //Visit v (post-order)
end procedure
```

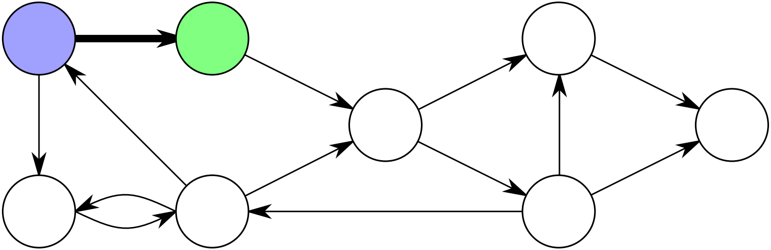
Depending on the application, some aspects (such as the status of edges) may be irrelevant.

DFS on Directed Graphs (1)



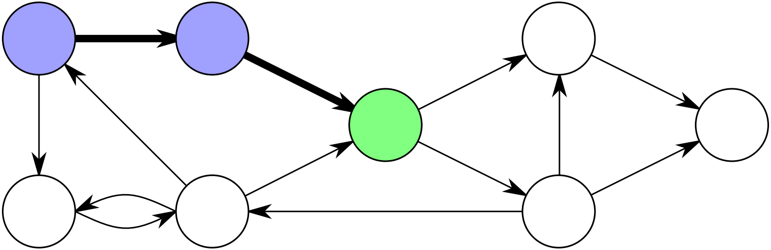
DFS has a natural generalization to directed graphs. The main difference is that the directions of edges must be respected.

DFS on Directed Graphs (2)



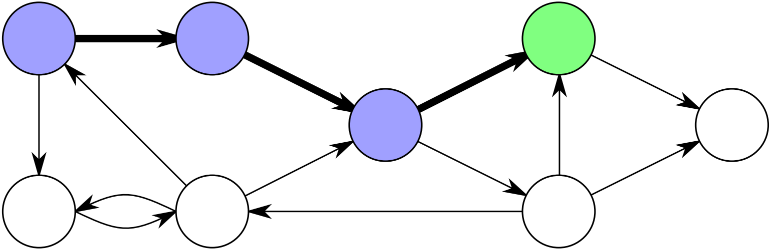
DFS has a natural generalization to directed graphs. The main difference is that the directions of edges must be respected.

DFS on Directed Graphs (3)



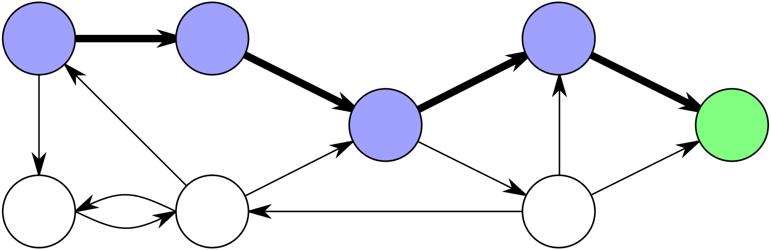
DFS has a natural generalization to directed graphs. The main difference is that the directions of edges must be respected.

DFS on Directed Graphs (4)



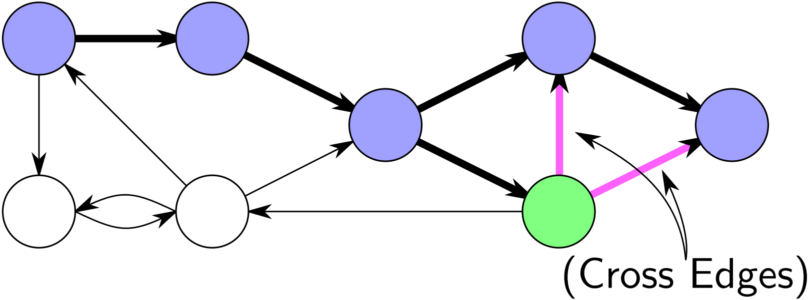
DFS has a natural generalization to directed graphs. The main difference is that the directions of edges must be respected.

DFS on Directed Graphs (5)



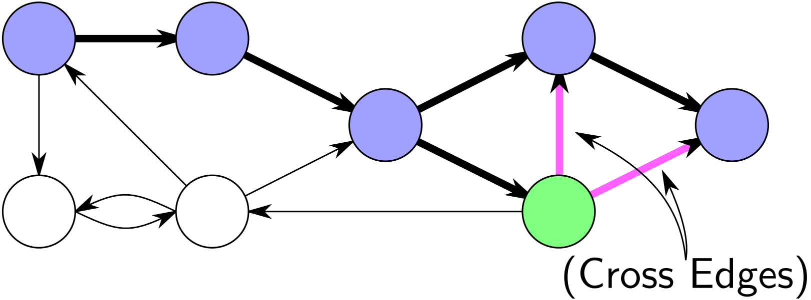
At this point, the current vertex has no unvisited neighbours, since the edges incident with the vertex are inbound edges.

DFS on Directed Graphs (6)



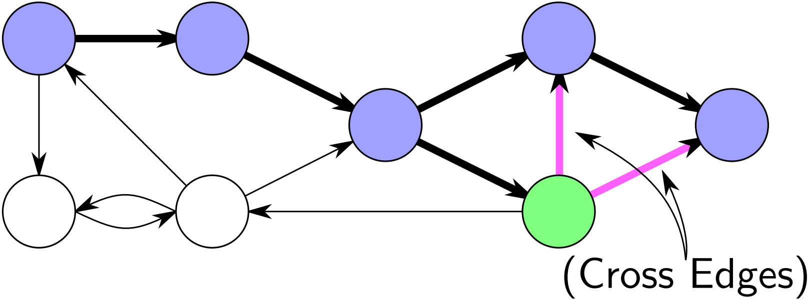
Recursion unwinds to the last vertex with unvisited neighbours.

DFS on Directed Graphs (7)



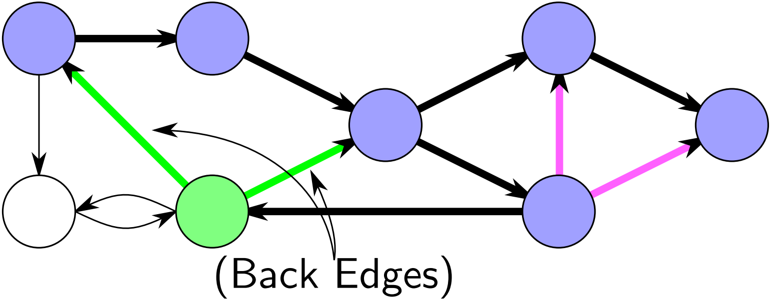
The neighbours of the current vertex are in a different branch of the DFS tree, so the edges are called **cross edges**.

DFS on Directed Graphs (8)

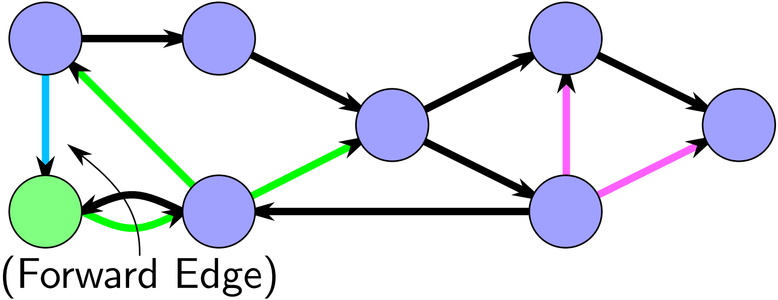


Exercise: Show that cross edges will never appear during DFS on an undirected graph.

DFS on Directed Graphs (9)

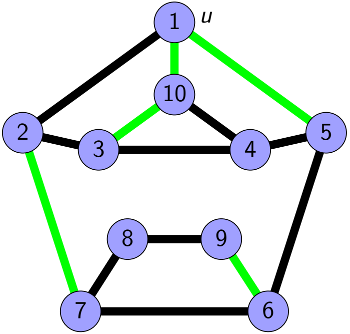


The two green edges are back edges, since the other endpoint of each edge is an ancestor of the current node in the DFS tree.

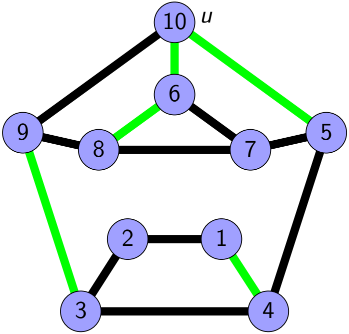


The light blue edge points from the root of the DFS tree to the current node. It is therefore a **forward edge**, since it points from a node to one of its descendants.

DFS Vertex Numbering (1)



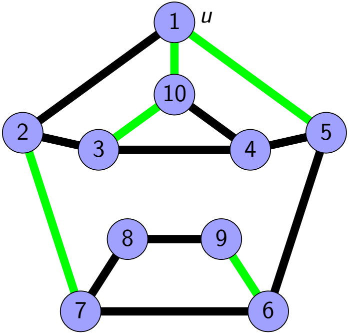
Pre-Order Numbering



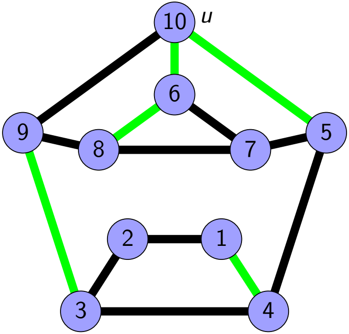
Post-Order Numbering

The order in which vertices are visited in a DFS traversal is often significant. Vertices can be assigned indices based on their order during a traversal.

DFS Vertex Numbering (2)



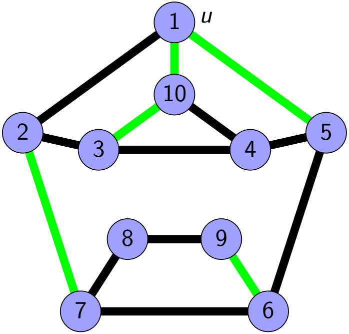
Pre-Order Numbering



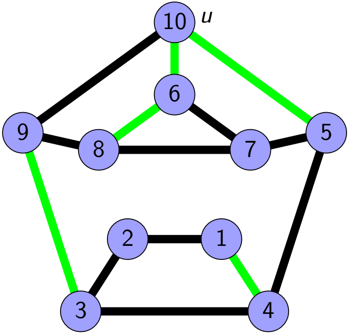
Post-Order Numbering

(In the diagrams above, the vertex u is the root of the DFS tree)

DFS Vertex Numbering (3)



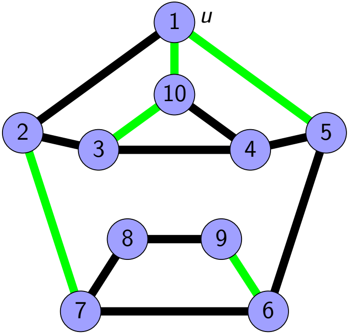
Pre-Order Numbering



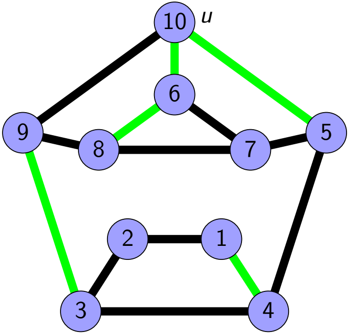
Post-Order Numbering

In a **pre-order** numbering, each vertex is assigned the next available index during the 'pre-order visit' (as DFS first arrives at the vertex).

DFS Vertex Numbering (4)



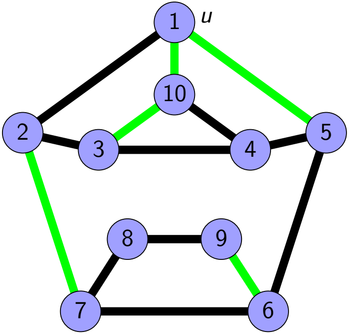
Pre-Order Numbering



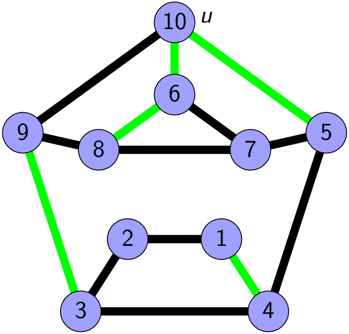
Post-Order Numbering

A pre-order numbering corresponds to the order in which vertices are visited in a pre-order traversal of the DFS tree.

DFS Vertex Numbering (5)



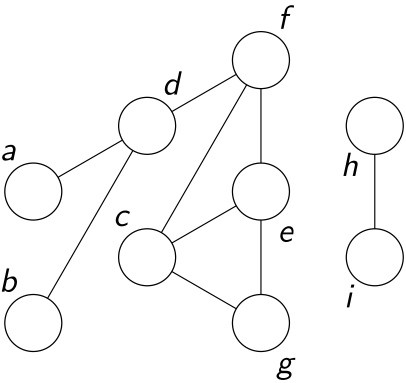
Pre-Order Numbering



Post-Order Numbering

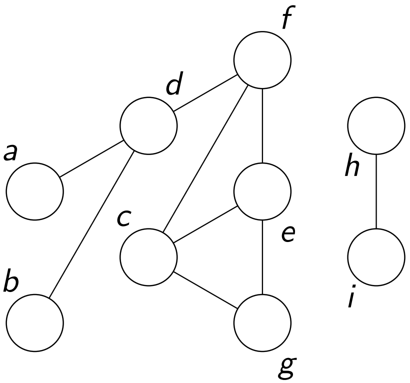
In a **post-order** numbering, each vertex is assigned the next available index during the ‘post-order visit’ (just before DFS leaves the vertex for the last time). A post-order numbering corresponds to a post-order traversal of the DFS tree.

Storing Traversal Trees (1)



Exercise: Perform a DFS traversal starting at vertex f above and store the resulting DFS tree.

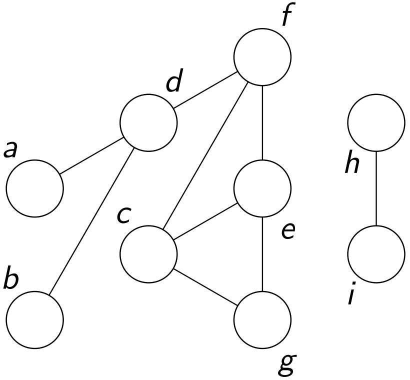
Storing Traversal Trees (2)



Vertex	a	b	c	d	e	f	g	h	i
Parent	×	×	×	×	×	×	×	×	×

Traversal trees (and other rooted trees) can be represented with a **parent array** structure. A parent array is an associative map which stores the parent of each vertex.

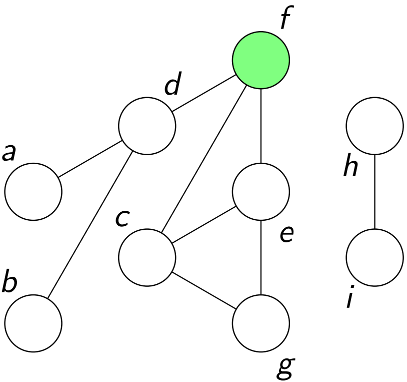
Storing Traversal Trees (3)



Vertex	a	b	c	d	e	f	g	h	i
Parent	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Vertices which are not part of the tree have no parent, and are marked as invalid (in this case, using the symbol $\times$). Before the traversal starts, all vertices have invalid parents.

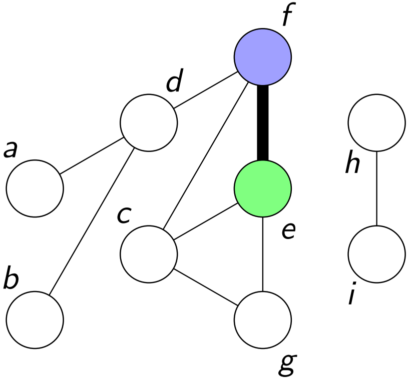
Storing Traversal Trees (4)



Vertex	a	b	c	d	e	f	g	h	i
Parent	×	×	×	×	×	f	×	×	×

At the beginning of the traversal, the starting vertex f is set to be its own parent (to indicate that f is the root).

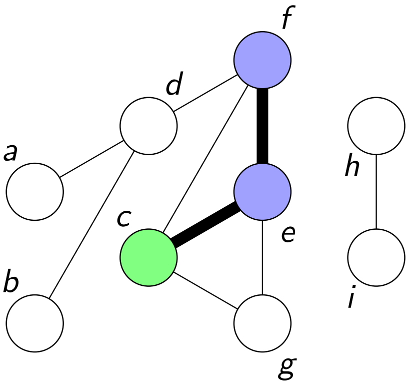
Storing Traversal Trees (5)



Vertex	a	b	c	d	e	f	g	h	i
Parent	×	×	×	×	f	f	×	×	×

Since vertex e is visited for the first time from vertex f, the parent of e is set to be f.

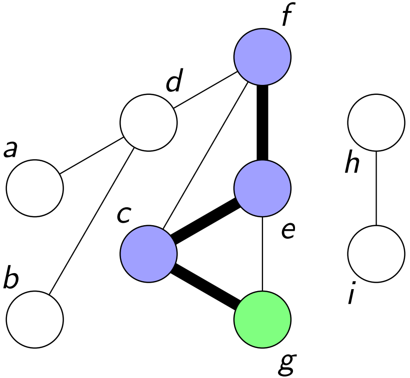
Storing Traversal Trees (6)



Vertex	a	b	c	d	e	f	g	h	i
Parent	×	×	e	×	f	f	×	×	×

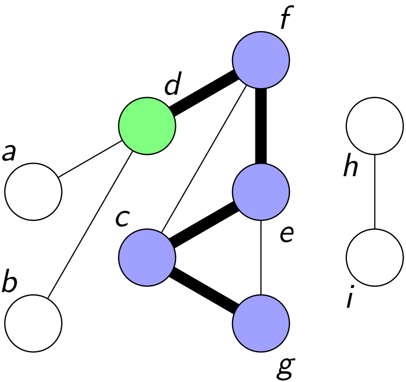
As the traversal proceeds, the parent of each newly discovered vertex w is set to be the vertex u on the other endpoint of its discovery edge.

Storing Traversal Trees (7)



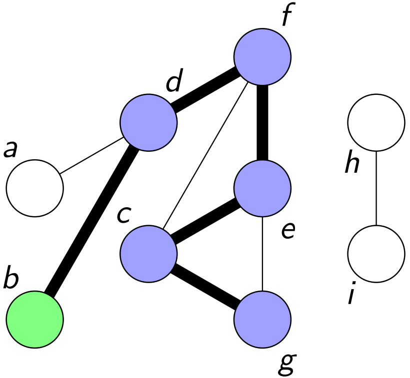
Vertex	a	b	c	d	e	f	g	h	i
Parent	×	×	e	×	f	f	c	×	×

Storing Traversal Trees (8)



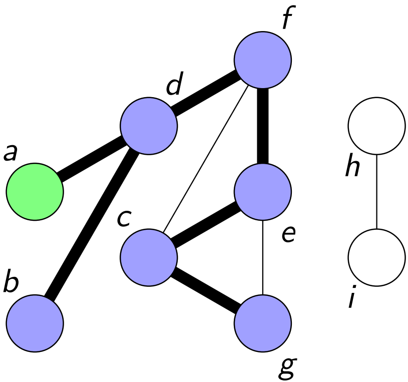
Vertex	a	b	c	d	e	f	g	h	i
Parent	×	×	e	f	f	f	c	×	×

Storing Traversal Trees (9)



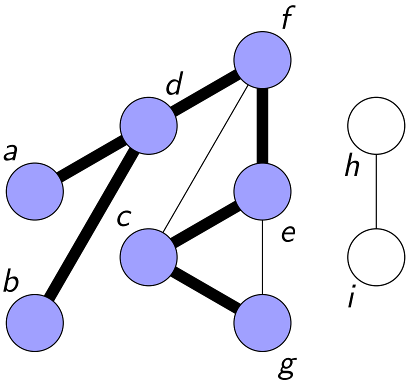
Vertex	a	b	c	d	e	f	g	h	i
Parent	×	d	e	f	f	f	c	×	×

Storing Traversal Trees (10)



Vertex	a	b	c	d	e	f	g	h	i
Parent	d	d	e	f	f	f	c	×	×

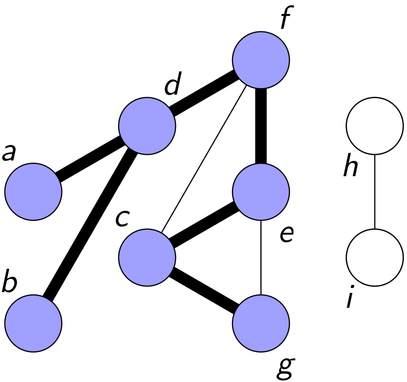
Storing Traversal Trees (11)



Vertex	a	b	c	d	e	f	g	h	i
Parent	d	d	e	f	f	f	c	×	×

When the traversal finishes, any vertices with an invalid parent (such as h and i in the above example) are not reachable from the starting vertex f.

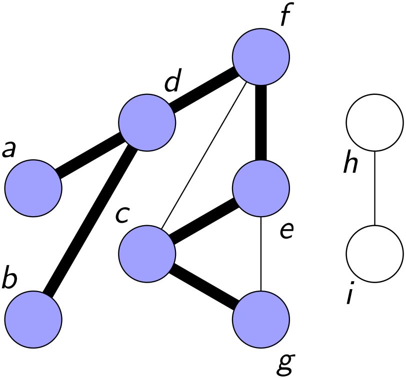
Storing Traversal Trees (12)



Vertex	a	b	c	d	e	f	g	h	i
Parent	d	d	e	f	f	f	c	×	×

If a vertex *v* has a valid parent, a path from *v* to the starting vertex can be found by following the parent pointers to the root of the tree.

Storing Traversal Trees (13)



Vertex	a	b	c	d	e	f	g	h	i
Parent	d	d	e	f	f	f	c	×	×

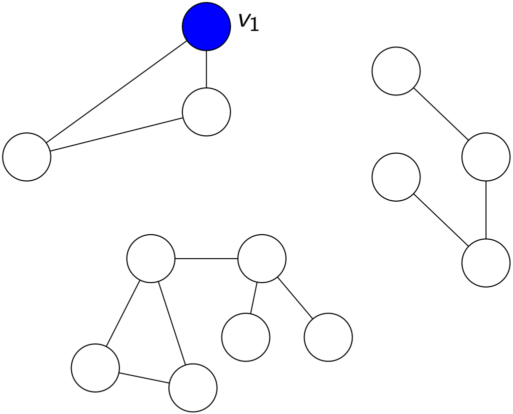
Observe that a single DFS traversal rooted at a vertex u will find a path from u to **all** vertices reachable from u.

Storing Traversal Trees (14)

```
procedure GenerateDFSTree(v)
    for each neighbour w of v do
        if Parent[w] = × then
            //(v, w) is a discovery edge
            Parent[w] = v
            GenerateDFSTree(w)
        end if
    end for
end procedure
```

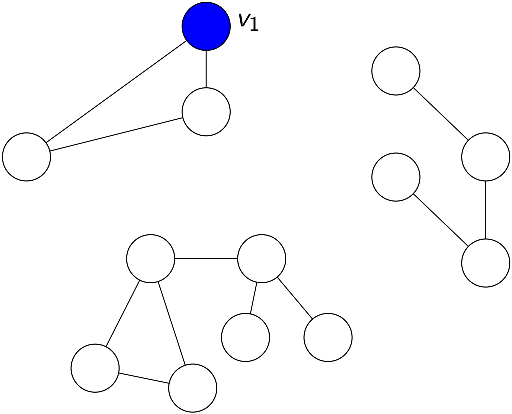
To start the recursive traversal above at a vertex u, set `Parent`$[u] = u$ and set all other elements of `Parent` to $\times$.

Finding Connected Components (1)



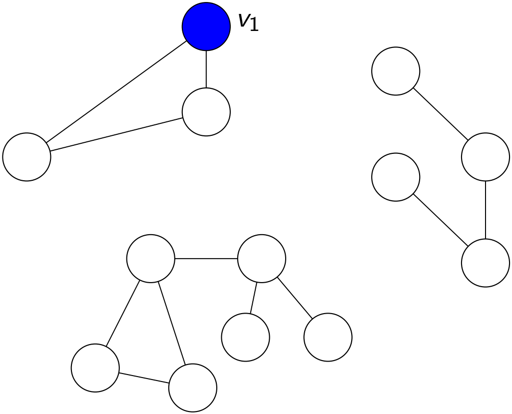
Exercise: Describe an algorithm to find the connected components of a graph G.

Finding Connected Components (2)



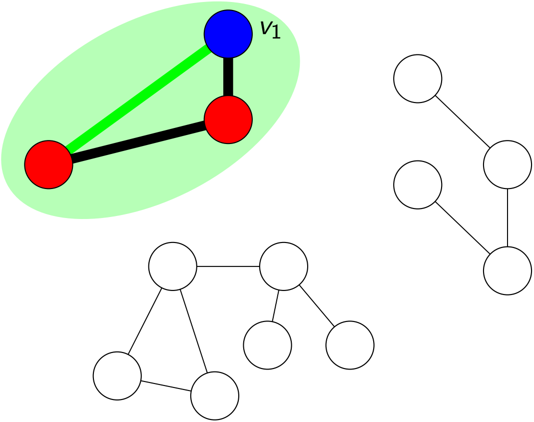
(This will be very helpful on Programming Assignment 3...)

Finding Connected Components (3)



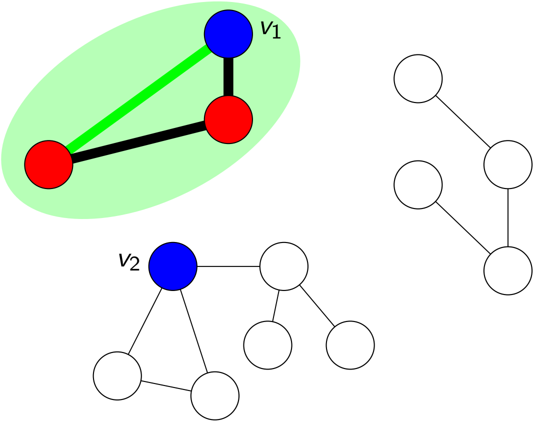
A DFS traversal rooted at a vertex v will visit all vertices reachable from v.

Finding Connected Components (4)



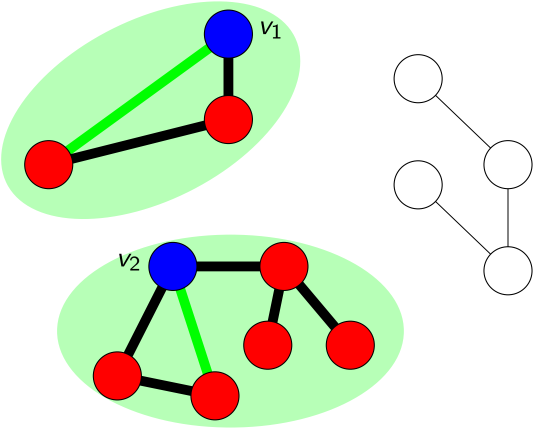
To find the components, choose a vertex v_1 and run DFS to find the component containing v_1.

Finding Connected Components (5)



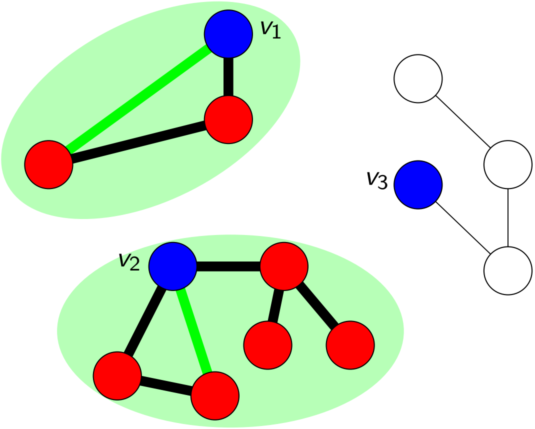
If all vertices in the graph have not been visited, choose another vertex v_2 and run DFS again.

Finding Connected Components (6)



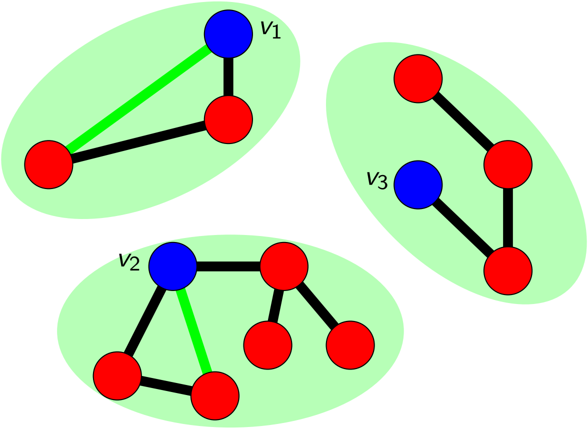
Each instance of DFS finds a spanning tree of one connected component.

Finding Connected Components (7)



While unvisited vertices remain, continue choosing an unvisited vertex and running DFS.

Finding Connected Components (8)



While unvisited vertices remain, continue choosing an unvisited vertex and running DFS.