CSC 225 - Summer 2019

Traversals II

Bill Bird

Department of Computer Science
University of Victoria

July 17, 2019

Iterative DFS (1)

```
procedure IterativeDFS(r)
    S ← Empty stack
    Parent ← Parent array, initialized to ×.
    Push r onto S.
    Parent[r] ← r
    while S is non-empty do
        v ← Peek(S)
        if all neighbours of v have been visited then
            Pop(S)
        else
            w ← an unvisited neighbour of v
            Parent[w] ← v
            Push w onto S
        end if
    end while
    return Parent
end procedure
```

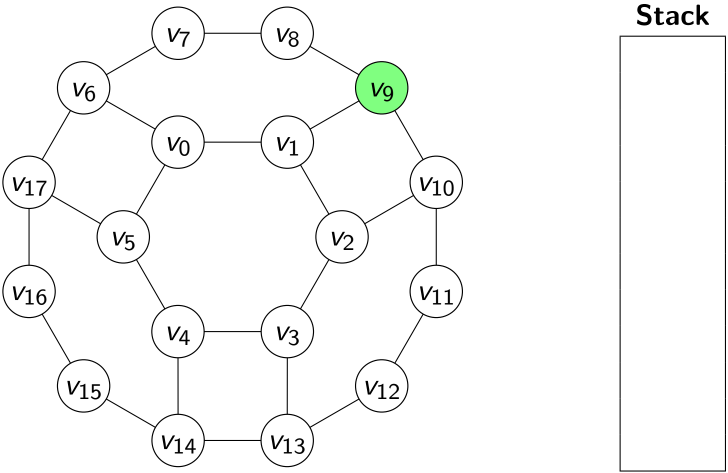
DFS can be implemented as an iterative algorithm using a stack.

Iterative DFS (2)

```
procedure ITERATIVEDFS(r)
    S ← Empty stack
    Parent ← Parent array, initialized to ×.
    Push r onto S.
    Parent[r] ← r
    while S is non-empty do
        v ← PEEK(S)
        if all neighbours of v have been visited then
            POP(S)
        else
            w ← an unvisited neighbour of v
            Parent[w] ← v
            Push w onto S
        end if
    end while
    return Parent
end procedure
```

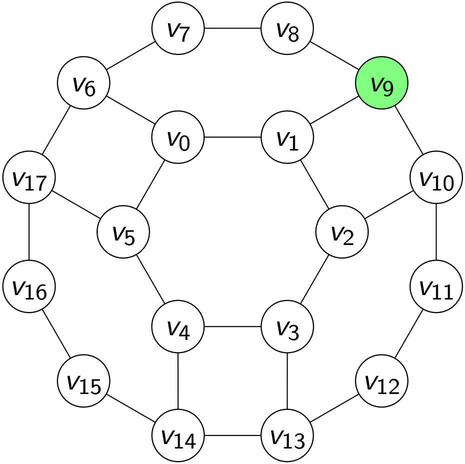
The pseudocode above takes the root vertex r and returns the parent array of the DFS tree.

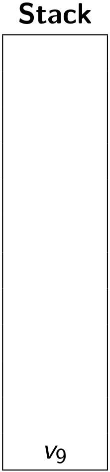
Iterative DFS (3)



Consider a DFS traversal using the stack based algorithm starting at vertex v_9 in the graph above.

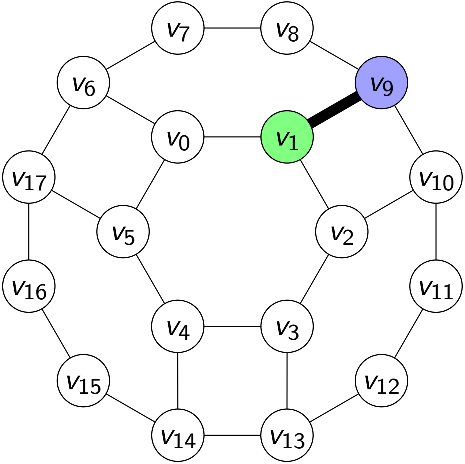
Iterative DFS (4)





First, push the root vertex v_9 onto the stack.

Iterative DFS (5)

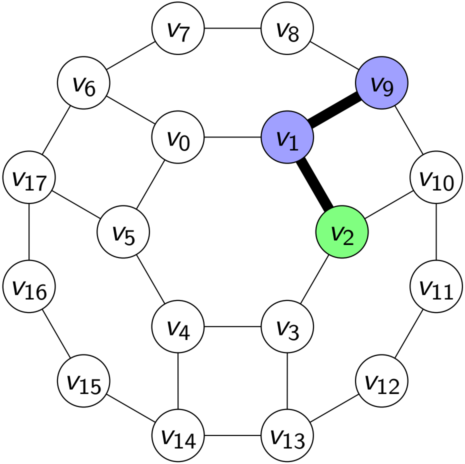


Stack

v_1
v_9

As each vertex is visited for the first time, it is added to the stack.

Iterative DFS (6)

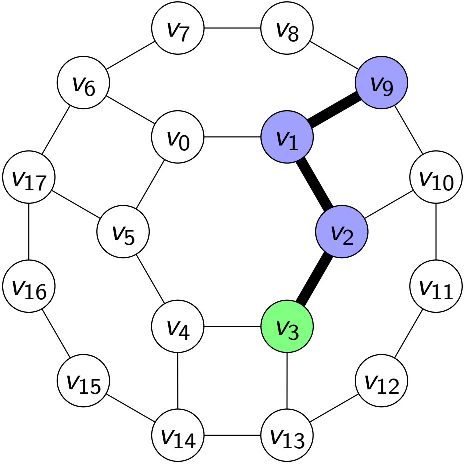


Stack

v_2
v_1
v_9

As each vertex is visited for the first time, it is added to the stack.

Iterative DFS (7)

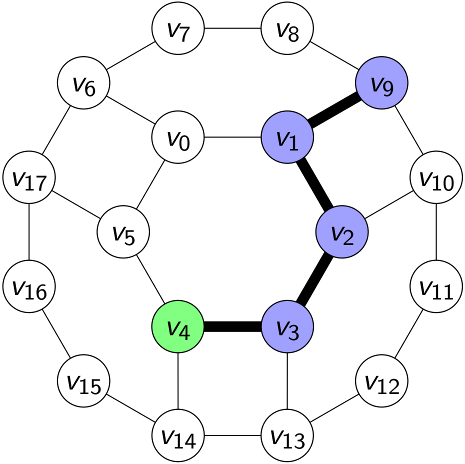


Stack

v_3
v_2
v_1
v_9

As each vertex is visited for the first time, it is added to the stack.

Iterative DFS (8)

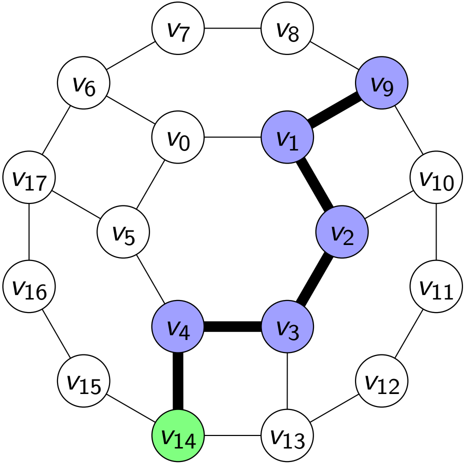


Stack

v_4
v_3
v_2
v_1
v_9

As each vertex is visited for the first time, it is added to the stack.

Iterative DFS (9)

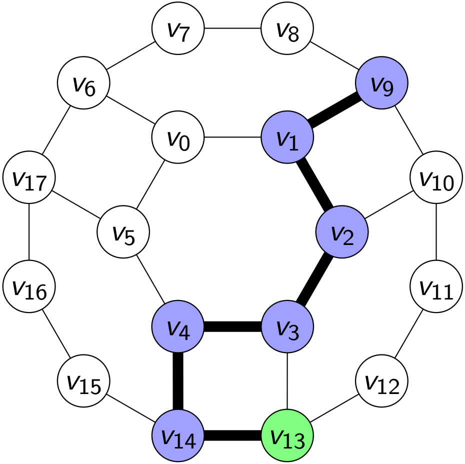


Stack

v_{14}
v_4
v_3
v_2
v_1
v_9

As each vertex is visited for the first time, it is added to the stack.

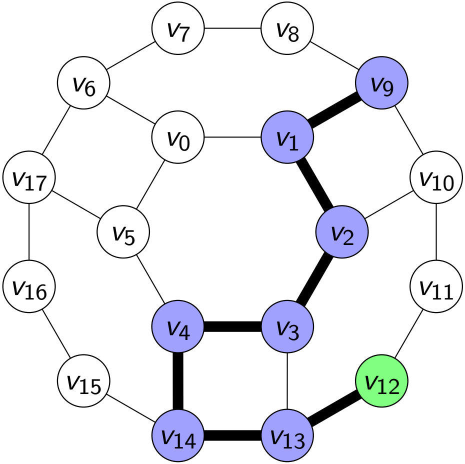
Iterative DFS (10)



Stack

v_{13}
v_{14}
v_4
v_3
v_2
v_1
v_9

As each vertex is visited for the first time, it is added to the stack.

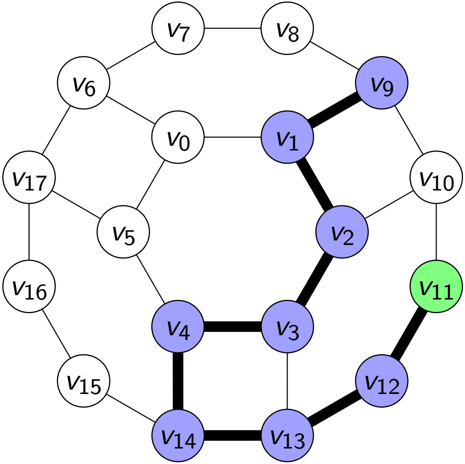


Stack

v_{12}
v_{13}
v_{14}
v_4
v_3
v_2
v_1
v_9

As each vertex is visited for the first time, it is added to the stack.

Iterative DFS (12)

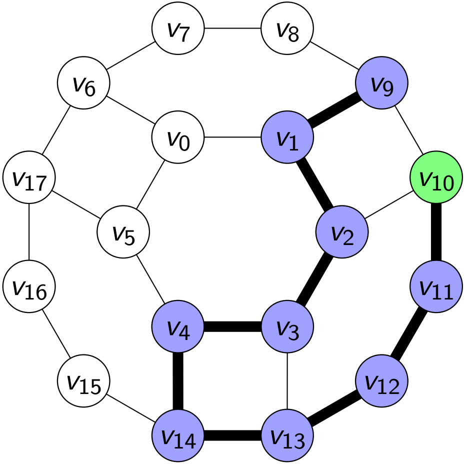


Stack

v_{11}
v_{12}
v_{13}
v_{14}
v_4
v_3
v_2
v_1
v_9

As each vertex is visited for the first time, it is added to the stack.

Iterative DFS (13)

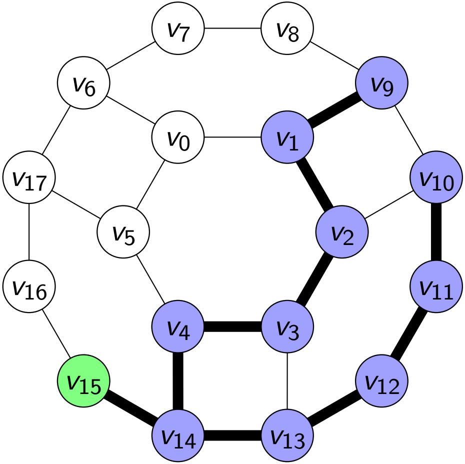


Stack

v_{10}
v_{11}
v_{12}
v_{13}
v_{14}
v_4
v_3
v_2
v_1
v_9

As each vertex is visited for the first time, it is added to the stack.

Iterative DFS (14)

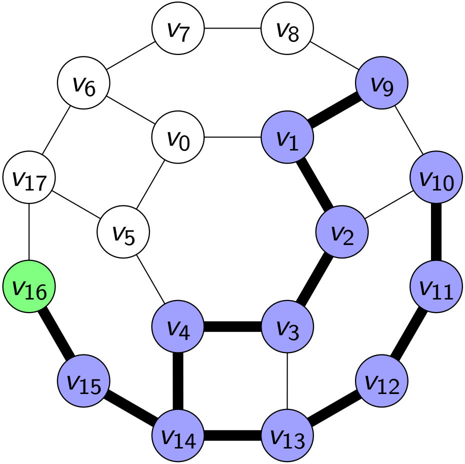


Stack

v_{15}
v_{14}
v_4
v_3
v_2
v_1
v_9

When the algorithm leaves a vertex for the last time, it is popped from the stack.

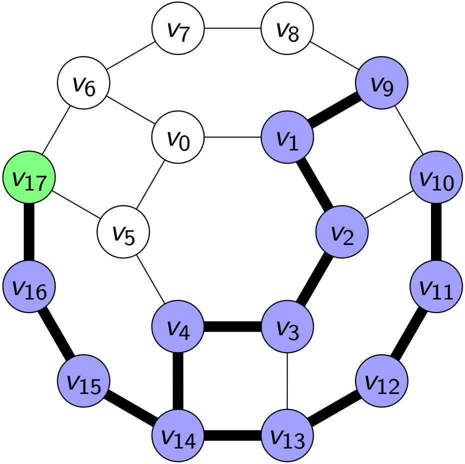
Iterative DFS (15)



Stack

v_{16}
v_{15}
v_{14}
v_4
v_3
v_2
v_1
v_9

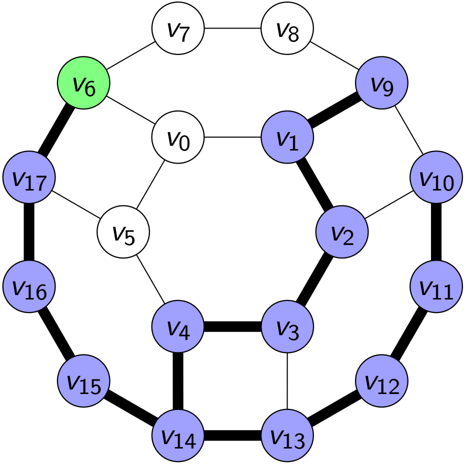
Iterative DFS (16)



Stack

v_{17}
v_{16}
v_{15}
v_{14}
v_4
v_3
v_2
v_1
v_9

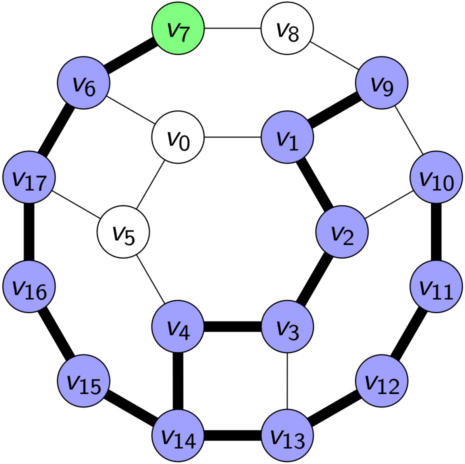
Iterative DFS (17)



Stack

v_6
v_{17}
v_{16}
v_{15}
v_{14}
v_4
v_3
v_2
v_1
v_9

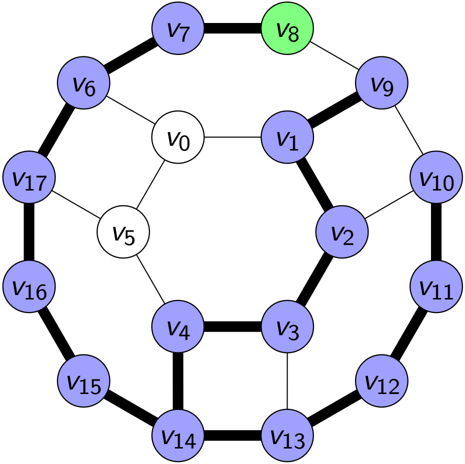
Iterative DFS (18)



Stack

v_7
v_6
v_{17}
v_{16}
v_{15}
v_{14}
v_4
v_3
v_2
v_1
v_9

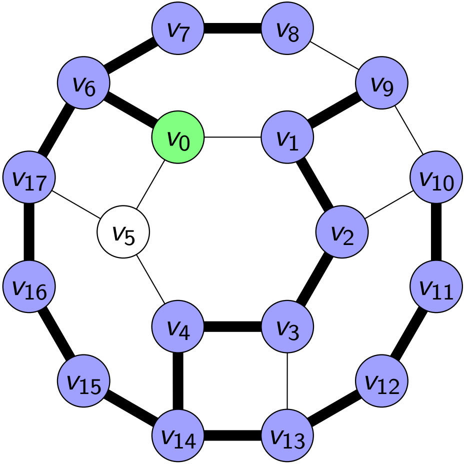
Iterative DFS (19)



Stack

v_8
v_7
v_6
v_{17}
v_{16}
v_{15}
v_{14}
v_4
v_3
v_2
v_1
v_9

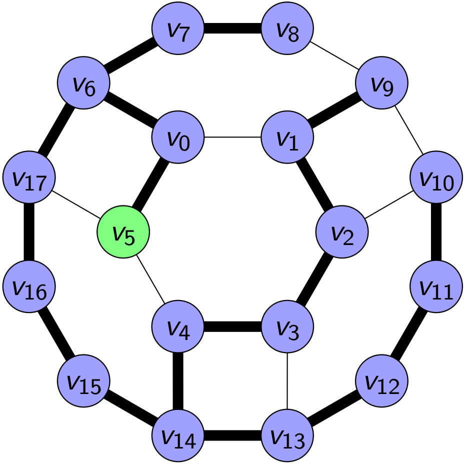
Iterative DFS (20)



Stack

v_0
v_6
v_{17}
v_{16}
v_{15}
v_{14}
v_4
v_3
v_2
v_1
v_9

Iterative DFS (21)



Stack
v_5
v_0
v_6
v_{17}
v_{16}
v_{15}
v_{14}
v_4
v_3
v_2
v_1
v_9

Iterative DFS (22)

```
procedure IterativeDFS(r)
    S ← Empty stack
    Parent ← Parent array, initialized to ×.
    Push r onto S.
    Parent[r] ← r
    while S is non-empty do
        v ← Peek(S)
        if all neighbours of v have been visited then
            Pop(S)
        else
            w ← an unvisited neighbour of v
            Parent[w] ← v
            Push w onto S
        end if
    end while
    return Parent
end procedure
```

Question: What if the traversal used a queue instead of a stack?

Breadth-First Search (1)

```
procedure BFS(r)
    Q ← Empty queue
    Enqueue r into Q
    Mark r as visited
    while Q is non-empty do
        v ← DEQUEUE(Q)
        for each neighbour w of v do
            if w is unvisited then
                Mark w as visited
                Enqueue w in Q
            end if
        end for
    end while
end procedure
```

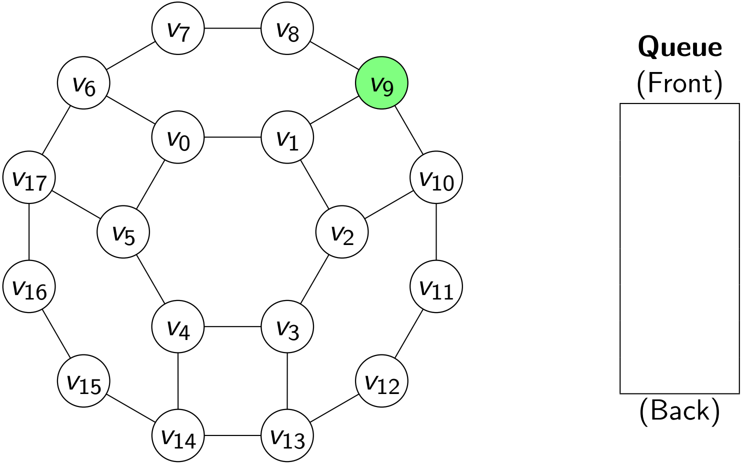
Using a queue produces a traversal called **breadth-first search** or **BFS**.

Breadth-First Search (2)

```
procedure GenerateBFSTree(r)
    Q ← Empty queue
    Parent ← Parent array, initialized to ×.
    Enqueue r into Q.
    Parent[r] ← r
    while Q is non-empty do
        v ← Dequeue(Q)
        for each neighbour w of v do
            if Parent[w] = × then
                Parent[w] ← v
                Enqueue w in Q
            end if
        end for
    end while
    return Parent
end procedure
```

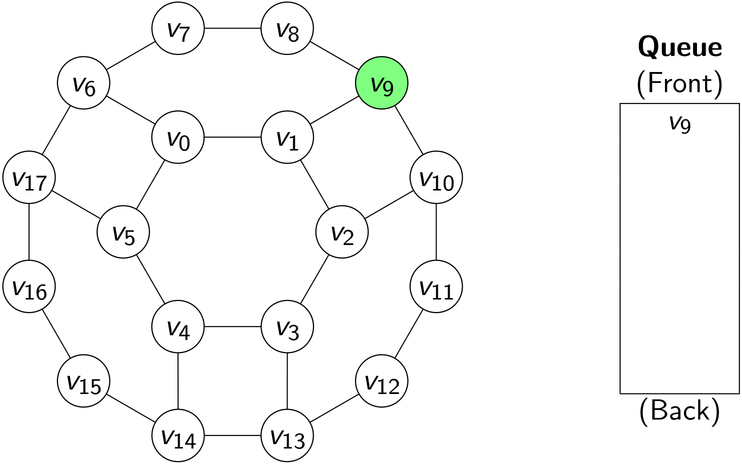
The pseudocode above generates and returns a BFS tree.

Breadth-First Search (3)



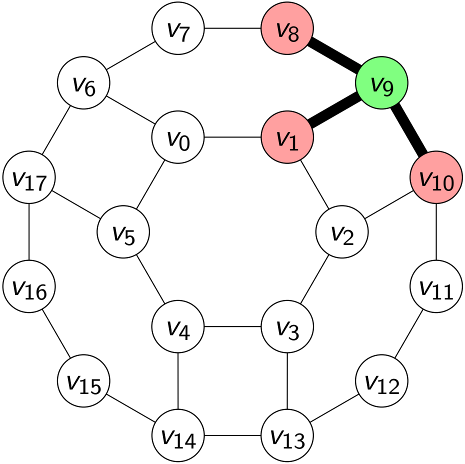
Consider a BFS traversal using the stack based algorithm starting at vertex v_9 in the graph above.

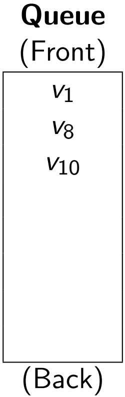
Breadth-First Search (4)



First, enqueue the root vertex v_9 in the queue.

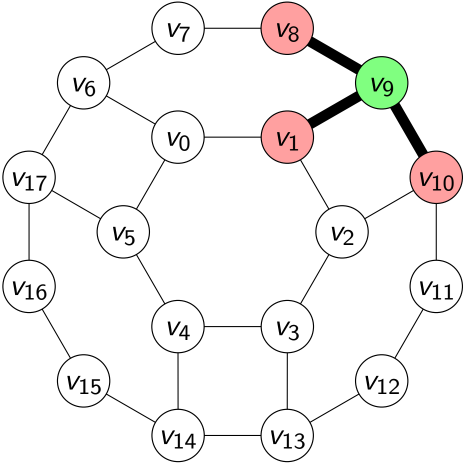
Breadth-First Search (5)

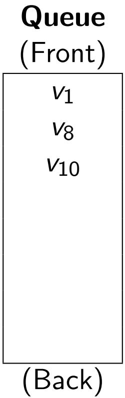




At each iteration of the main loop, the vertex of the front of the queue is removed and all of its unvisited neighbours are added.

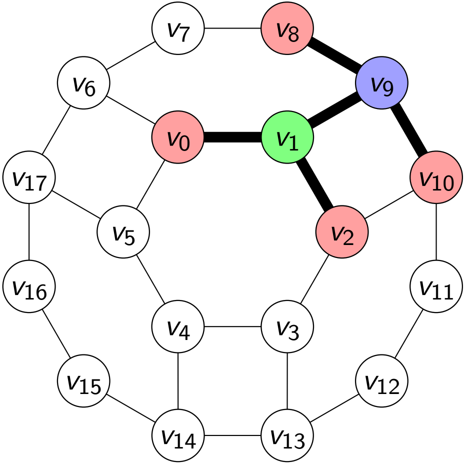
Breadth-First Search (6)





(Vertices which are in the queue but have not been visited are shown in red).

Breadth-First Search (7)

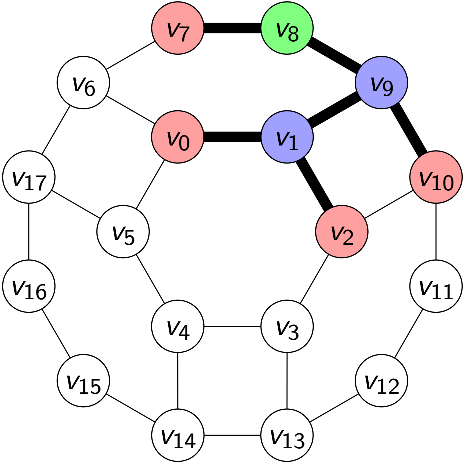


Queue
(Front)

v_8
v_{10}
v_0
v_2

(Back)

Breadth-First Search (8)

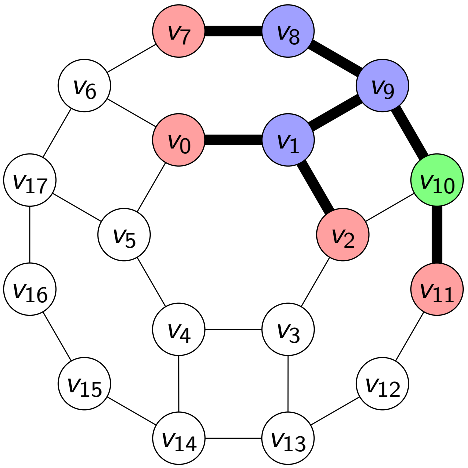


Queue
(Front)

v_{10}
v_0
v_2
v_7

(Back)

Breadth-First Search (9)

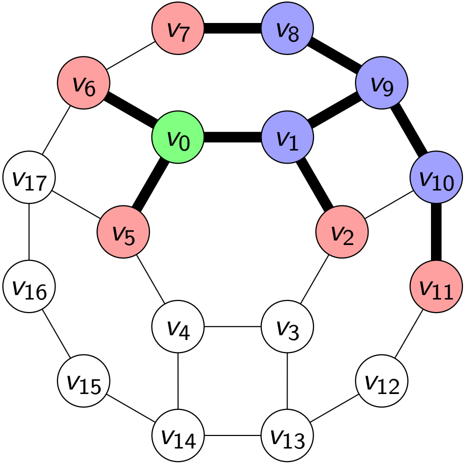


Queue
(Front)

v_0
v_2
v_7
v_{11}

(Back)

Breadth-First Search (10)

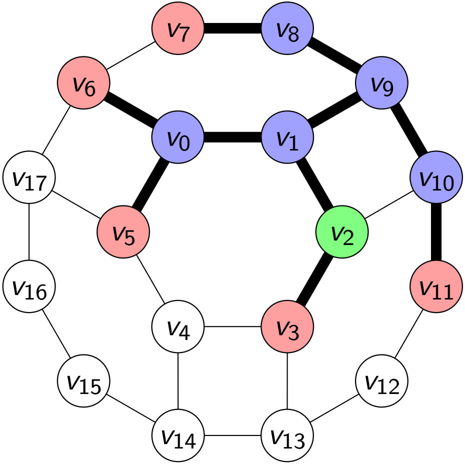


Queue
(Front)

v_2
v_7
v_{11}
v_5
v_6

(Back)

Breadth-First Search (11)



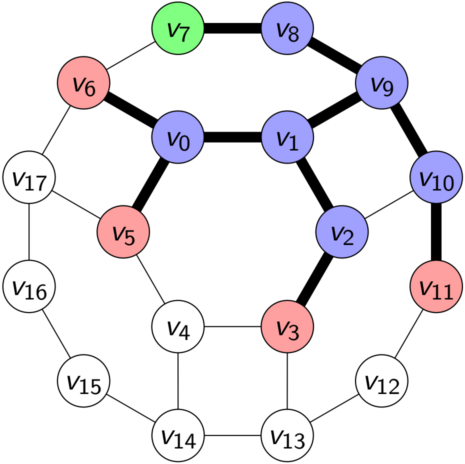
Queue

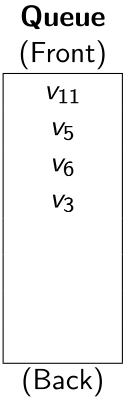
(Front)

v_7
v_{11}
v_5
v_6
v_3

(Back)

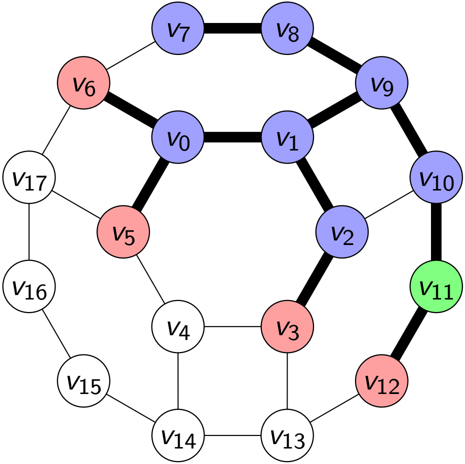
Breadth-First Search (12)





Even though v_6 has not been visited yet, it has already been added to the BFS tree (and the queue).

Breadth-First Search (13)

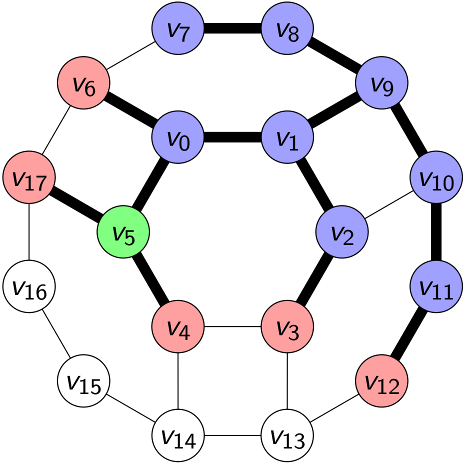


Queue
(Front)

v_5
v_6
v_3
v_{12}

(Back)

Breadth-First Search (14)

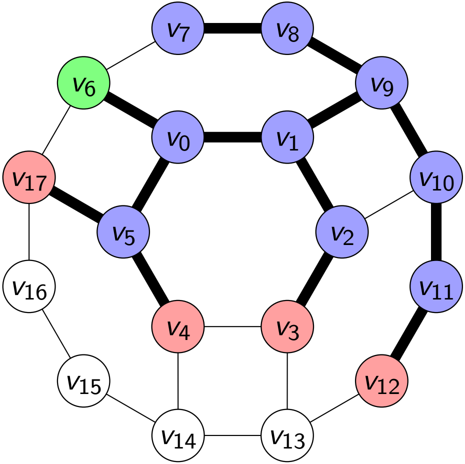


Queue
(Front)

v_6
v_3
v_{12}
v_4
v_{17}

(Back)

Breadth-First Search (15)

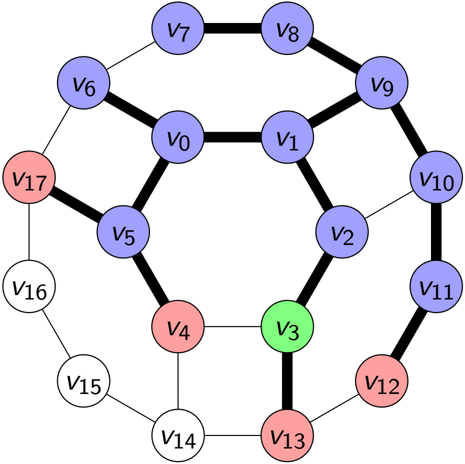


Queue
(Front)

v_3
v_{12}
v_4
v_{17}

(Back)

Breadth-First Search (16)

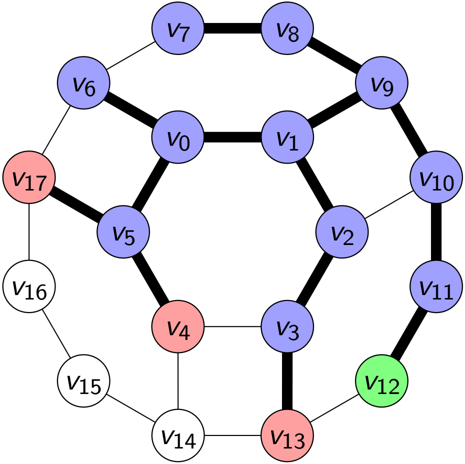


Queue
(Front)

v_{12}
v_4
v_{17}
v_{13}

(Back)

Breadth-First Search (17)

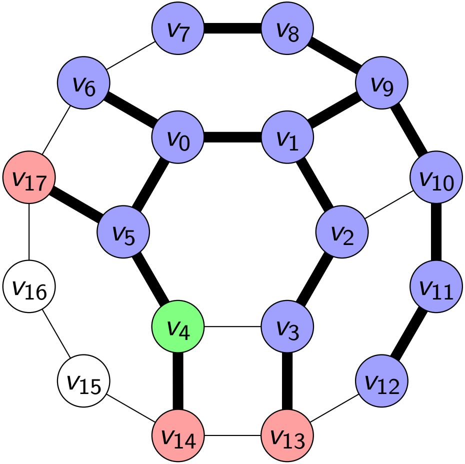


Queue
(Front)

v_4
v_{17}
v_{13}

(Back)

Breadth-First Search (18)

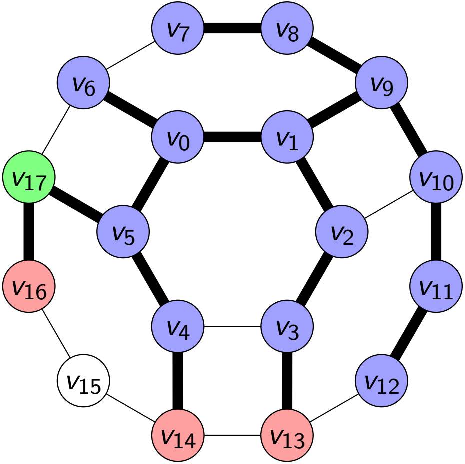


Queue
(Front)

v_{17}
v_{13}
v_{14}

(Back)

Breadth-First Search (19)

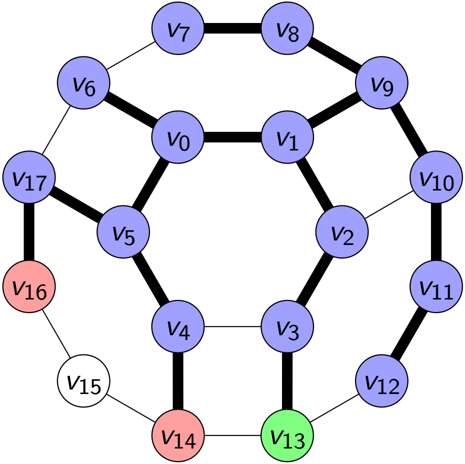


Queue
(Front)

v_{13}
v_{14}
v_{16}

(Back)

Breadth-First Search (20)

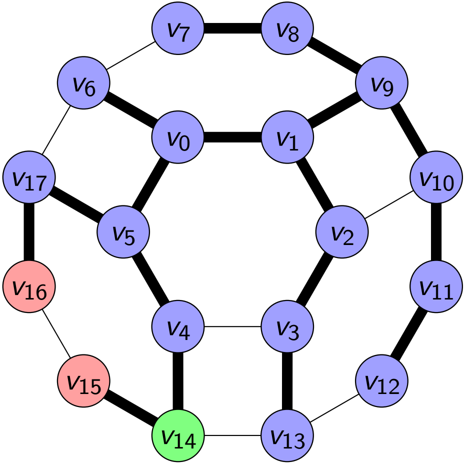


Queue
(Front)

v_{14}
v_{16}

(Back)

Breadth-First Search (21)

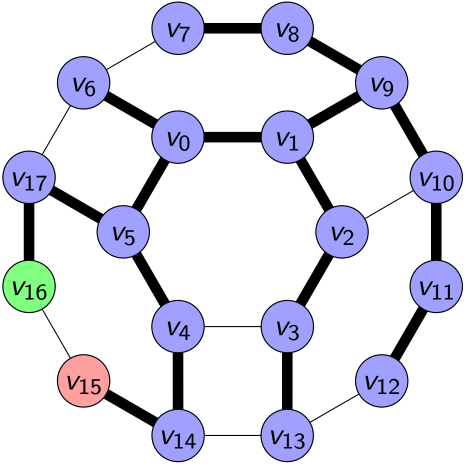


Queue
(Front)

v_{16}
v_{15}

(Back)

Breadth-First Search (22)

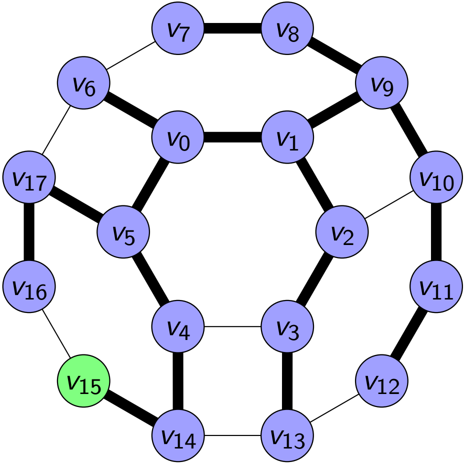


Queue
(Front)

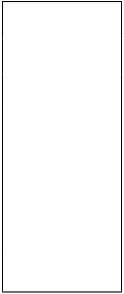
v_{15}

(Back)

Breadth-First Search (23)

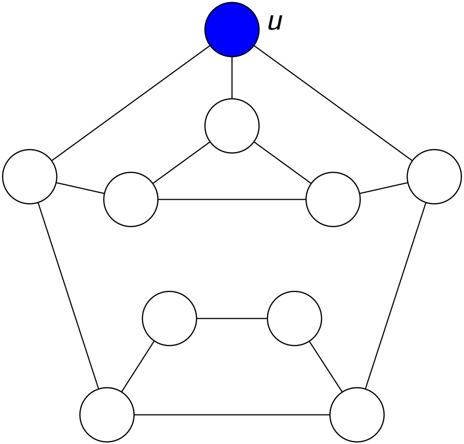


Queue
(Front)



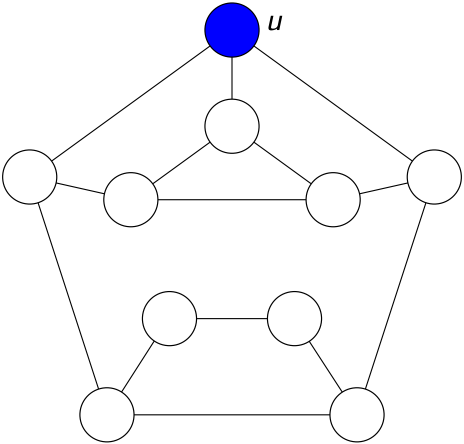
(Back)

BFS and BFS Trees (1)



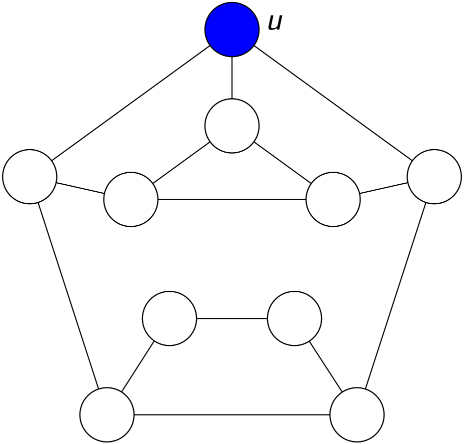
BFS is a more ‘conservative’ traversal and spreads out slowly from the starting vertex.

BFS and BFS Trees (2)



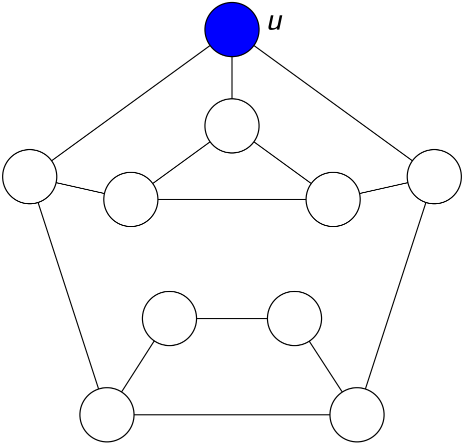
As a result, BFS trees have several valuable properties that DFS trees lack.

BFS and BFS Trees (3)



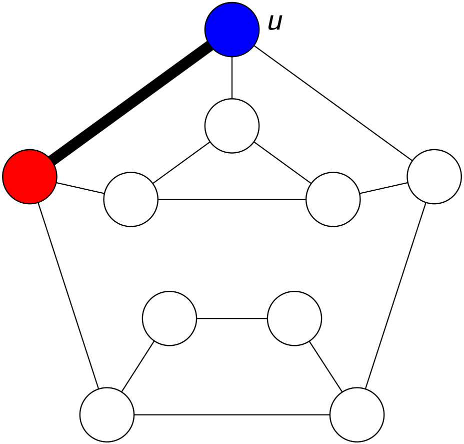
In particular, the path from any node to the root in a BFS tree is guaranteed to be the shortest path possible.

BFS and BFS Trees (4)



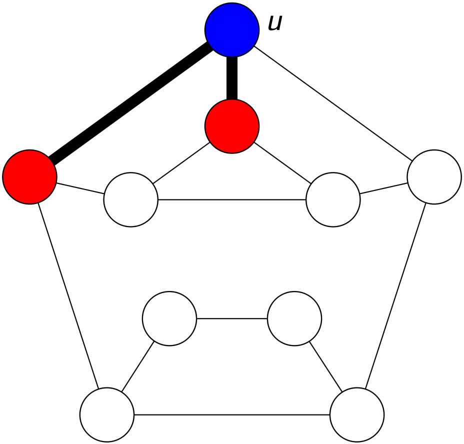
The vertices in level i of the BFS tree are the vertices with minimum distance i from the root of the tree. The root vertex is at level 0.

BFS and BFS Trees (5)



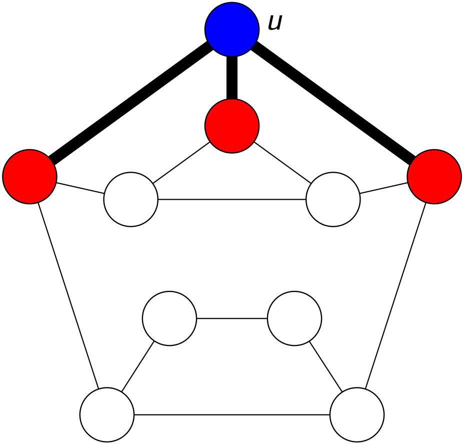
The vertices adjacent to the root are at distance 1.

BFS and BFS Trees (6)



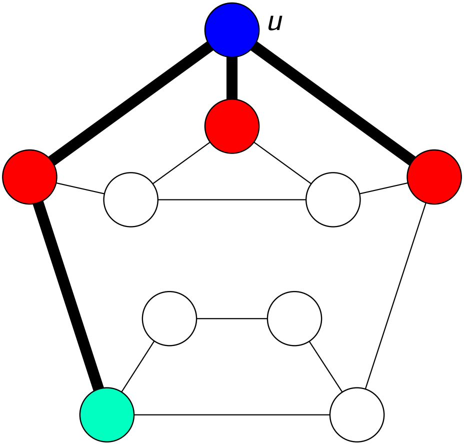
The vertices adjacent to the root are at distance 1.

BFS and BFS Trees (7)



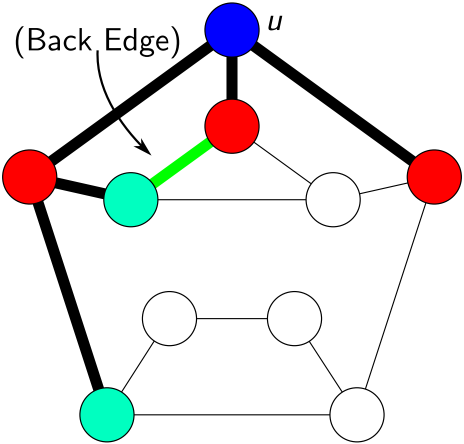
The vertices adjacent to the root are at distance 1.

BFS and BFS Trees (8)



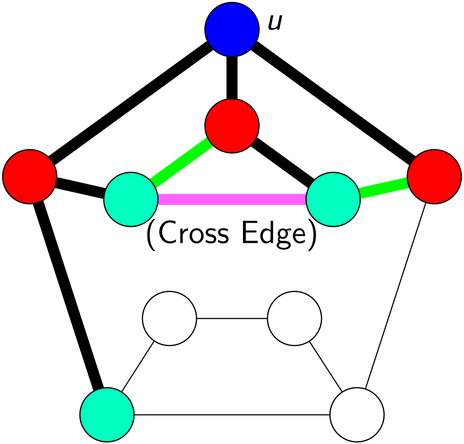
The neighbours of vertices at level 1 have distance 2.

BFS and BFS Trees (9)



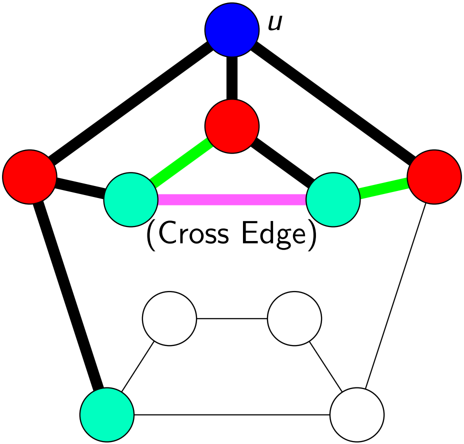
Non-tree edges between vertices on different levels are **back edges** (as in DFS).

BFS and BFS Trees (10)



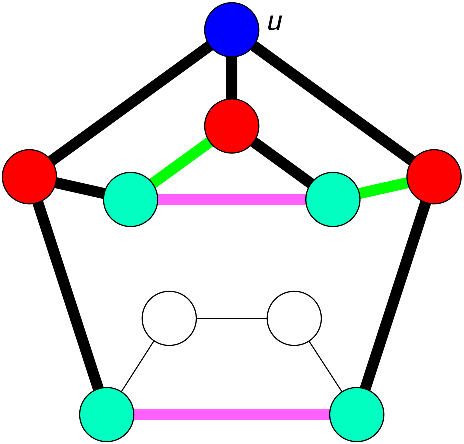
Non-tree edges between vertices on the same level are **cross edges**.

BFS and BFS Trees (11)

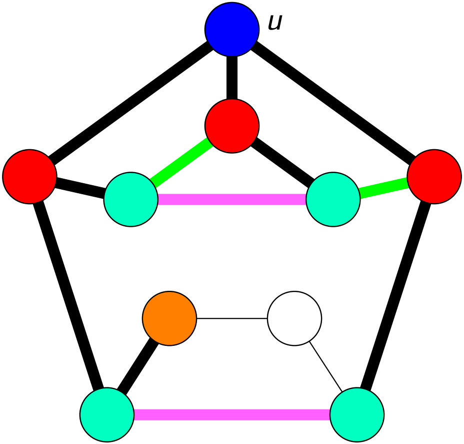


BFS can produce cross edges in undirected graphs, unlike DFS (which only produces cross edges in directed graphs).

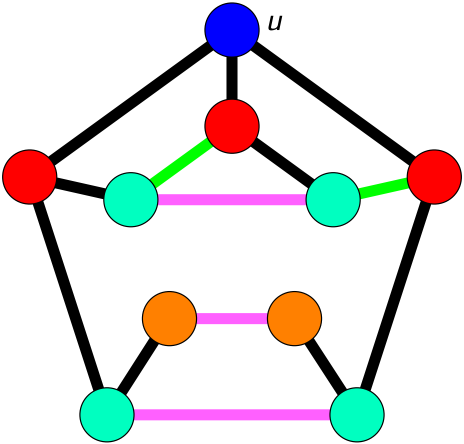
BFS and BFS Trees (12)



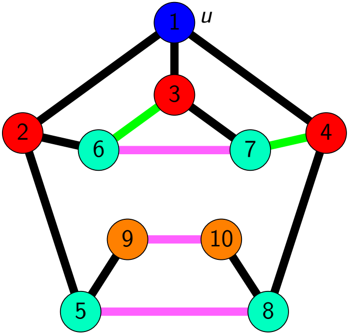
BFS and BFS Trees (13)



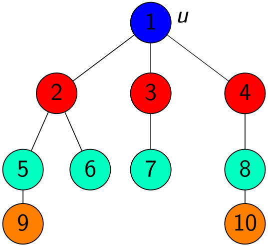
BFS and BFS Trees (14)



BFS and BFS Trees (15)



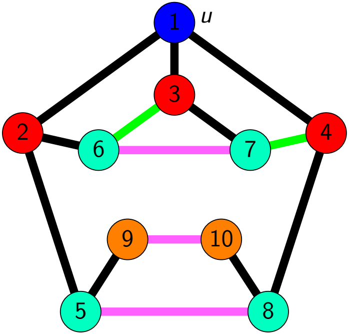
Vertex Numbering



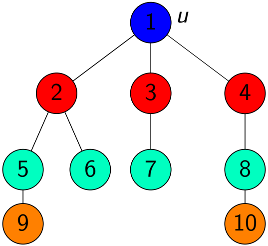
BFS Tree

The ordering of vertices in a BFS is equivalent to a level-order traversal of the underlying spanning tree.

BFS and BFS Trees (16)



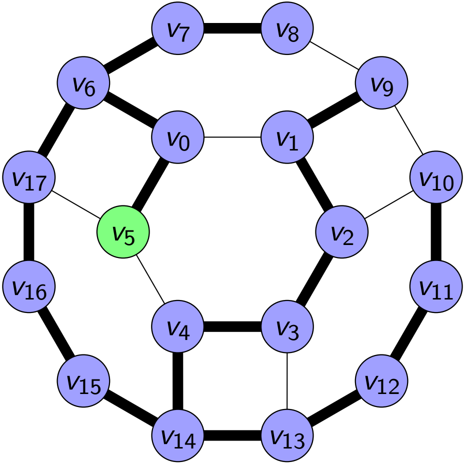
Vertex Numbering



BFS Tree

Note that paths from each node w to the root u are minimum length u-w paths in G.

Running Time of Traversals (1)



Question: What is the running time of DFS or BFS on a graph with n vertices and m edges?

Running Time of Traversals (2)

In both DFS and BFS, each vertex is visited and processed at most once.

When a vertex v is visited, the traversal algorithms iterate over all neighbours of v and traverse any unvisited neighbours. The total running time is then

$$n + \sum_{v \in V(G)} \deg(v) = n + 2m$$

Therefore, the running time of DFS and BFS is

$$\Theta(n + m)$$