Jan 2021 | © Guven

Depth First Search (DFS) Algorithm

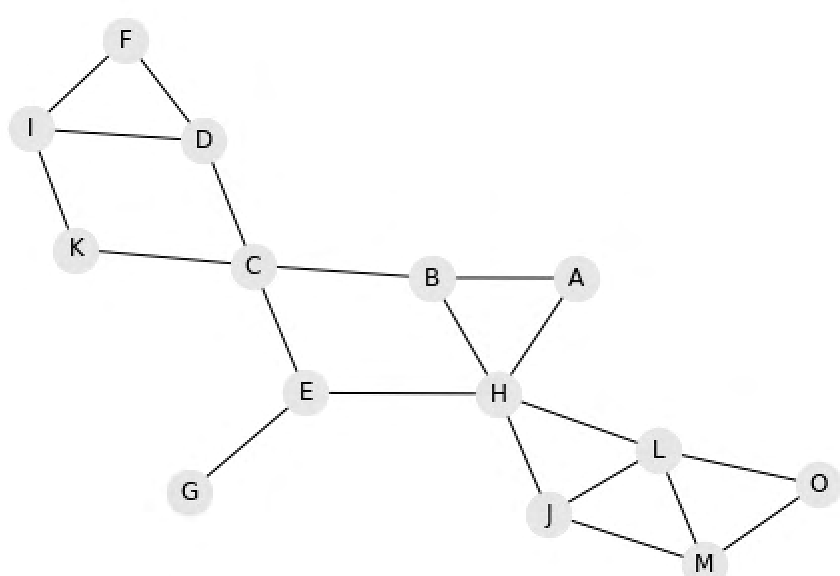
This notebook demonstrates,

1. Depth First Search
2. Breadth First Search
3. An example with 14 nodes
4. Uses `networkx` library to draw the graphs

In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import networkx as nx
print(f'networkx version {nx.__version__}')

def draw_graph(_g, _pos):
    nx.draw(_g, _pos, node_size=500, node_color='0.9', with_labels=True)

Edges = [('A','B'),('A','H'),('B','C'),('B','H'),('C','D'),('C','E'),('D','F'),('D','I'),('C','K'),
         ('E','G'),('E','H'),('F','I'),('H','J'),('H','L'),('I','K'),('J','L'),('L','M'),
         ('L','O'), ('J','M'), ('M','O')]

G = nx.Graph()
G.add_edges_from(Edges)

draw_graph(G, nx.kamada_kawai_layout(G))
```

```
networkx version 2.4
```



The edges are provided as the input with 2-tuples of 2 labels which correspond to node names.
The following example code will convert the input to an adjacency list,
i.e. a dictionary with keys as vertex labels and values as the **Vertex class** which has **label**, **dfs** and **parent** fields.
DFS will start from a given root and then **traverse** the graph.

Note: DFS uses an edge list, i.e. dictionary of lists where the key is the node (i.e. return of the `adj_list`).
`draw_tree` uses list of 2-tuples for edges (i.e. `Edges`).

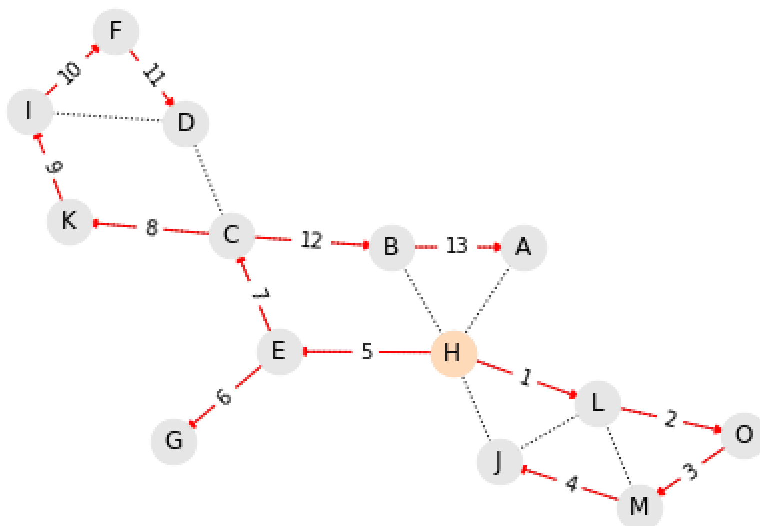
In [2]:
```
# Set a vertex class to store DFS (first timestamp) and parent
class Vertex:
    def __init__(self, label):
        self.label=label; self.dfs=None; self.p=None; self.seen=False

# Root is specially handled
ROOT_PARENT_LABEL = 'nul'
ROOT_PARENT = Vertex(ROOT_PARENT_LABEL)

# Generate an adjacency list from the input edges _e
def adj_list(_e:list) -> (dict,dict):  # _e is a list of 2-tuples
    from collections import defaultdict
    assert type(_e) is list
    assert type(_e[0]) is tuple

    edges = defaultdict(list)
    vertices = {}  # convert the labels to vertex objects
    for v1, v2 in _e:
        if v1 not in vertices:
            vertices[v1] = Vertex(v1)
        if v2 not in vertices:
            vertices[v2] = Vertex(v2)
        edges[v1] += [vertices[v2]]
        edges[v2] += [vertices[v1]]
    # Sort adjacency list edges
    for v in edges:
        edges[v] = sorted(edges[v], key=lambda x:x.label)
    #
    return vertices, edges

# DFS uses a stack iteratively to avoid recursion, pre-order traversa
l
# edge list key is the vertex label
def dfs_stack(_e, _root):
    # pre-order traversal to populate dfs values
    dfscounter = 1
    stack = [_root]  # stack is simply a Python list
    while len(stack) > 0:
        v1 = stack.pop()
        if not v1.seen:  # not visited yet
            v1.seen = True
            v1.dfs = dfscounter
            dfscounter += 1
            # edge dictionary key is vertex label, value is list of n
odes
            for v2 in _e[v1.label]:
                if not v2.seen:  # not visited yet
                    v2.p = v1  # set parent
                    stack += [v2]
```

Draw the tree generated by the DFS algorithm.

In [3]:
```
def draw_tree(_vertices:dict, _edges:list) -> None:
    # g will show entire graph, back-edges dotted
    g = nx.Graph()
    g.add_edges_from(_edges)
    pos = nx.kamada_kawai_layout(g)  # node positions, shared among a
ll graphs
    # tree edges
    e2, edgelabel = [], {}
    for v in _vertices.values():
        if v.p.label == ROOT_PARENT_LABEL:  # handle root as a specia
l case
            root = [v.label]
            continue
        #
        e2 += [(v.p.label, v.label)]
        edgelabel[(v.p.label, v.label)] = str(v.dfs-1)

    g_di = nx.DiGraph()
    g_di.add_edges_from(e2)
    # also show back-edges
    nx.draw(g, pos, node_size=500, node_color='0.9', with_labels=True
, style='dotted')
    nx.draw(g, pos, node_size=500, node_color='peachpuff', nodelist=r
oot, edgelist=[])
    nx.draw_networkx_edges(g_di, pos, edge_color='r')
    nx.draw_networkx_edge_labels(g_di, pos, edgelabel)
```

First example.

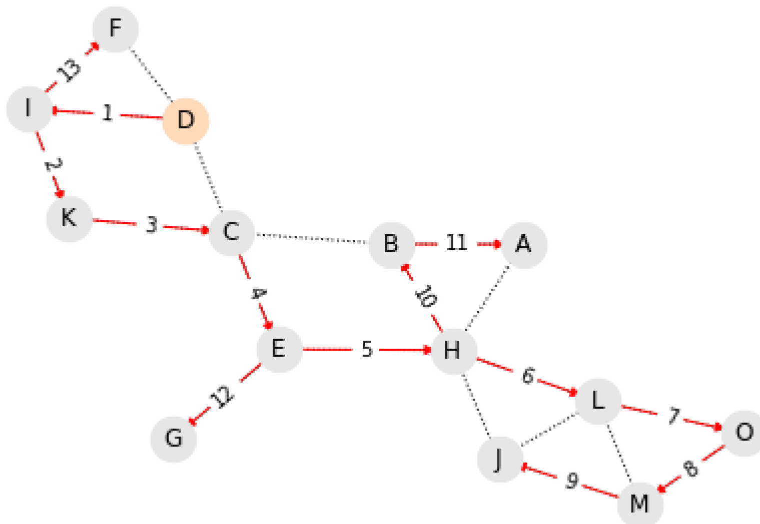
In [4]:
```
# DFS expects vertex objects
vertices1, edges1 = adj_list(Edges)

# Set the root
root = vertices1['H']
root.p = ROOT_PARENT

dfs_stack(edges1, root)

draw_tree(vertices1, Edges)
```



Now change the root, run the script again to observe how the DFS algorithm works.

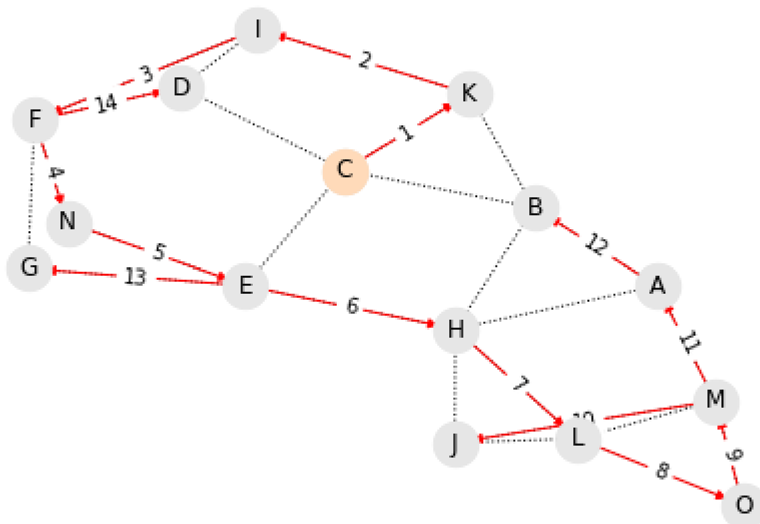
In [5]:

```
vertices2, edges2 = adj_list(Edges)

root = vertices2['D']
root.p = ROOT_PARENT

dfs_stack(edges2, root)
draw_tree(vertices2, Edges)
```



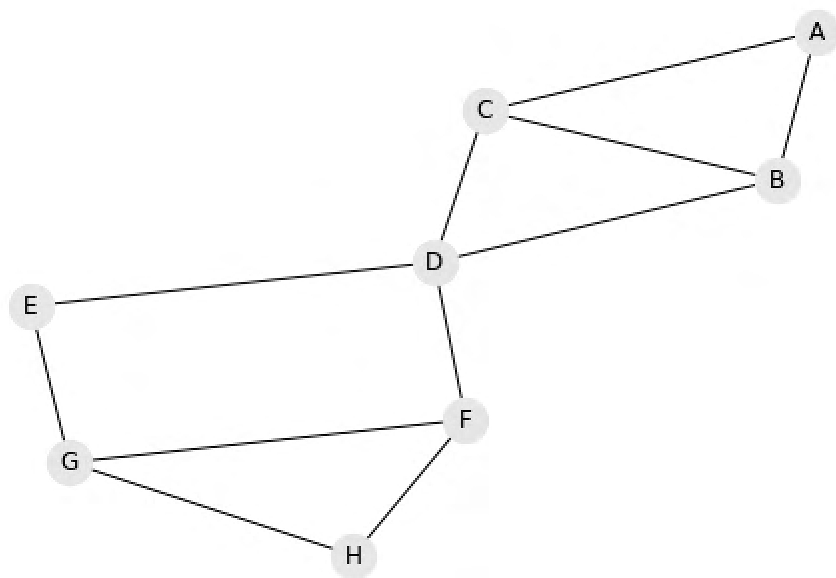
A bigger graph.

In [6]:
```
Edges2 = Edges + [('F','N'),('N','E'),('F','G'),('A','M'),('K','B')]

vertices3, edges3 = adj_list(Edges2)

root = vertices3['C']
root.p = ROOT_PARENT

dfs_stack(edges3, root)
draw_tree(vertices3, Edges2)
```

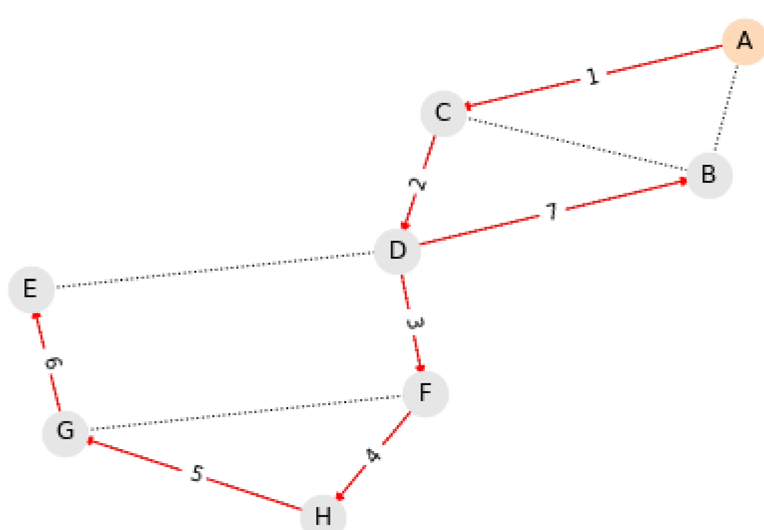


Exercise: Try the above code for higher number of nodes and verify the algorithm actually works.

In [7]:
```python
Edges10 = [('A','B'),('B','C'),('B','D'),('C','A'),('C','D'),('D','E'),('D','F'),
           ('E','G'),('F','G'),('F','H'),('H','G')]

G = nx.Graph()
G.add_edges_from(Edges10)

draw_graph(G, nx.kamada_kawai_layout(G))
```



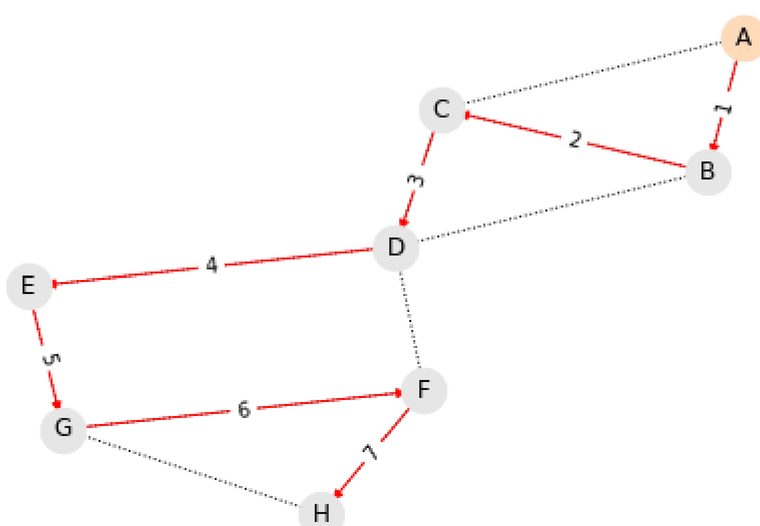
In [8]:
```python
vertices1, edges1 = adj_list(Edges10)

# Set the root
root = vertices1['A']
root.p = ROOT_PARENT

dfs_stack(edges1, root)

draw_tree(vertices1, Edges10)
```



- Let's try recursion.

In [9]:
```
Dfscounter = 1

def dfs_recursion(_e, v1):
    global Dfscounter
    # pre-order traversal to populate dfs values
    print(v1.label)  # Print the order of vertices visited
    v1.seen = True
    v1.dfs = Dfscounter
    Dfscounter += 1
    # edge dictionary key is vertex label, value is list of nodes
    for v2 in _e[v1.label]:
        if v2.seen == False:
            v2.p = v1  # set parent
            dfs_recursion(_e, v2)


vertices1, edges1 = adj_list(Edges10)

# Set the root
root = vertices1['A']
root.p = ROOT_PARENT

dfs_recursion(edges1, root)

draw_tree(vertices1, Edges10)
```

```
A
B
C
D
E
G
F
H
```



- Let's try BFS.

In [10]:
```
# BFS uses a queue iteratively to avoid recursion, in-order traversal
# edge list key is the vertex label
def bfs(_e, _root):
    # in-order traversal to populate dfs values
    dfscounter = 1
    queue = [_root]  # queue is simply a Python list
    while len(queue) > 0:
        v1 = queue.pop(0)  # Get the first element
        if not v1.seen:  # not visited yet
            v1.seen = True
            v1.dfs = dfscounter
            dfscounter += 1
            # edge dictionary key is vertex label, value is list of n
odes
            for v2 in _e[v1.label]:
                if not v2.seen:  # not visited yet
                    v2.p = v1  # set parent
                    queue += [v2]

vertices1, edges1 = adj_list(Edges10)

# Set the root
root = vertices1['A']
root.p = ROOT_PARENT

bfs(edges1, root)

draw_tree(vertices1, Edges10)
```