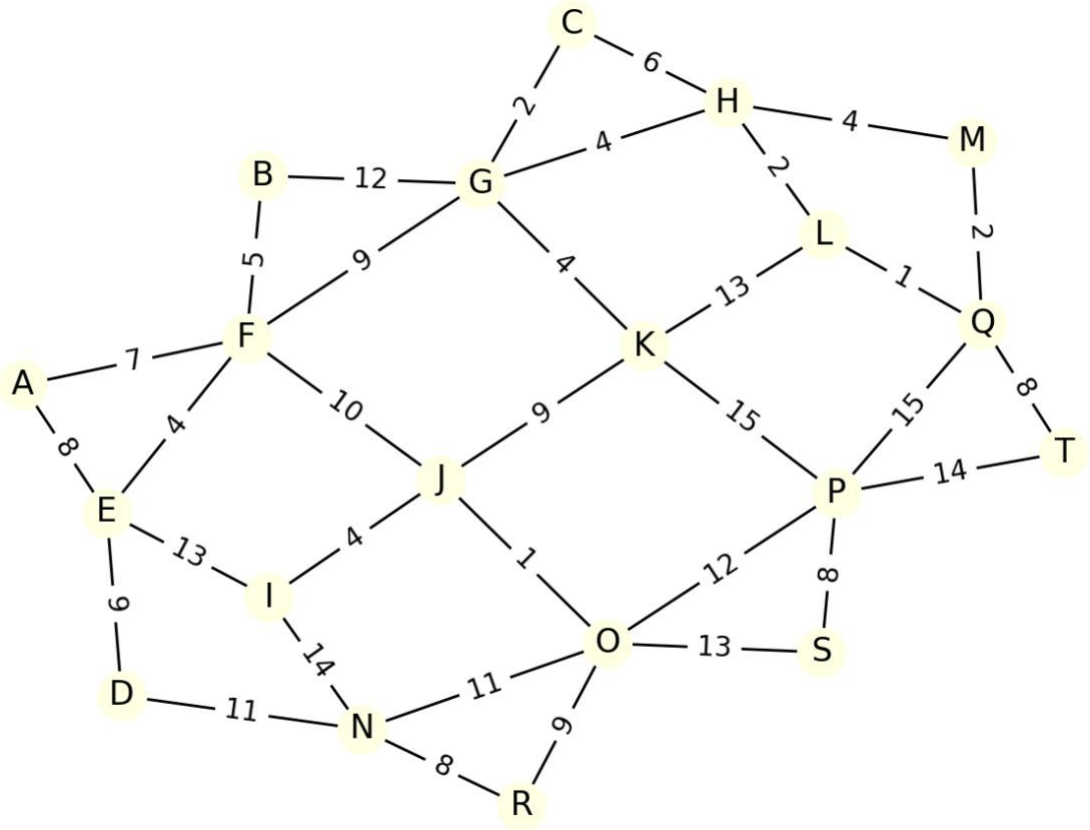
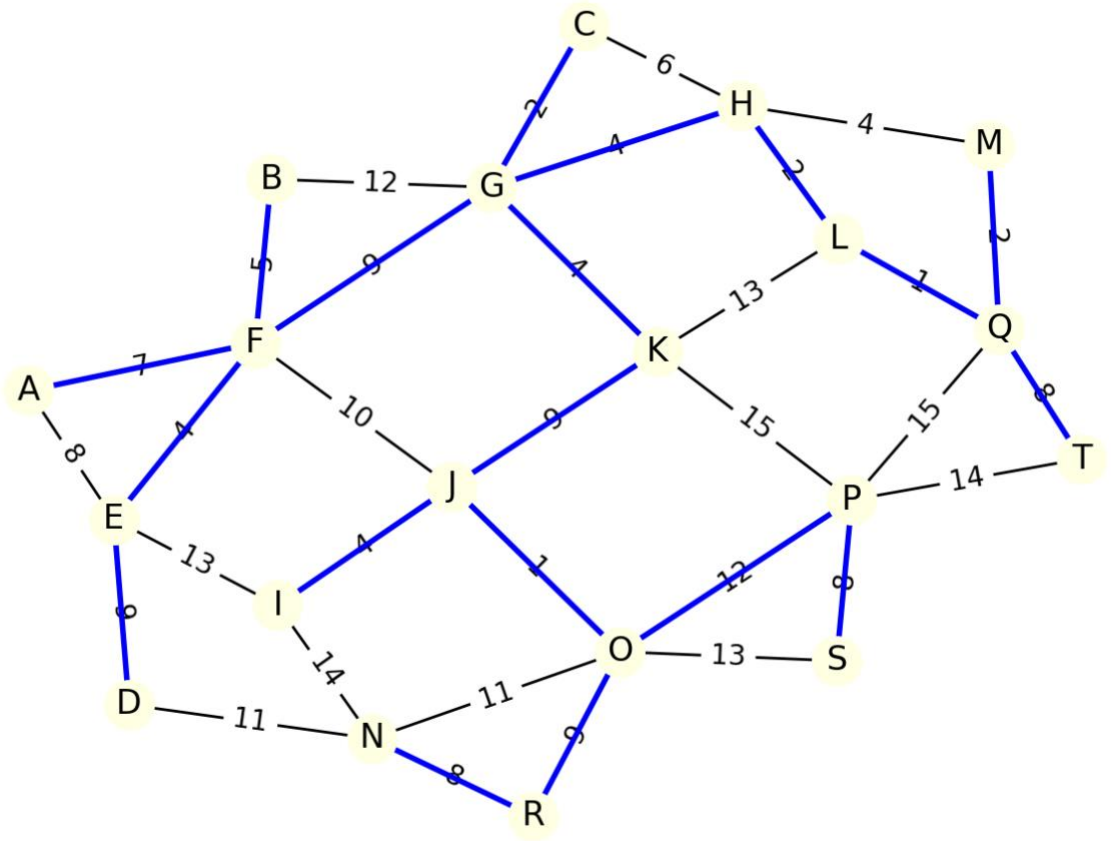
1. Connected, undirected graph of 20 nodes with weights



2. Prim's algorithm output. The input is the graph from number 1



Associated Code

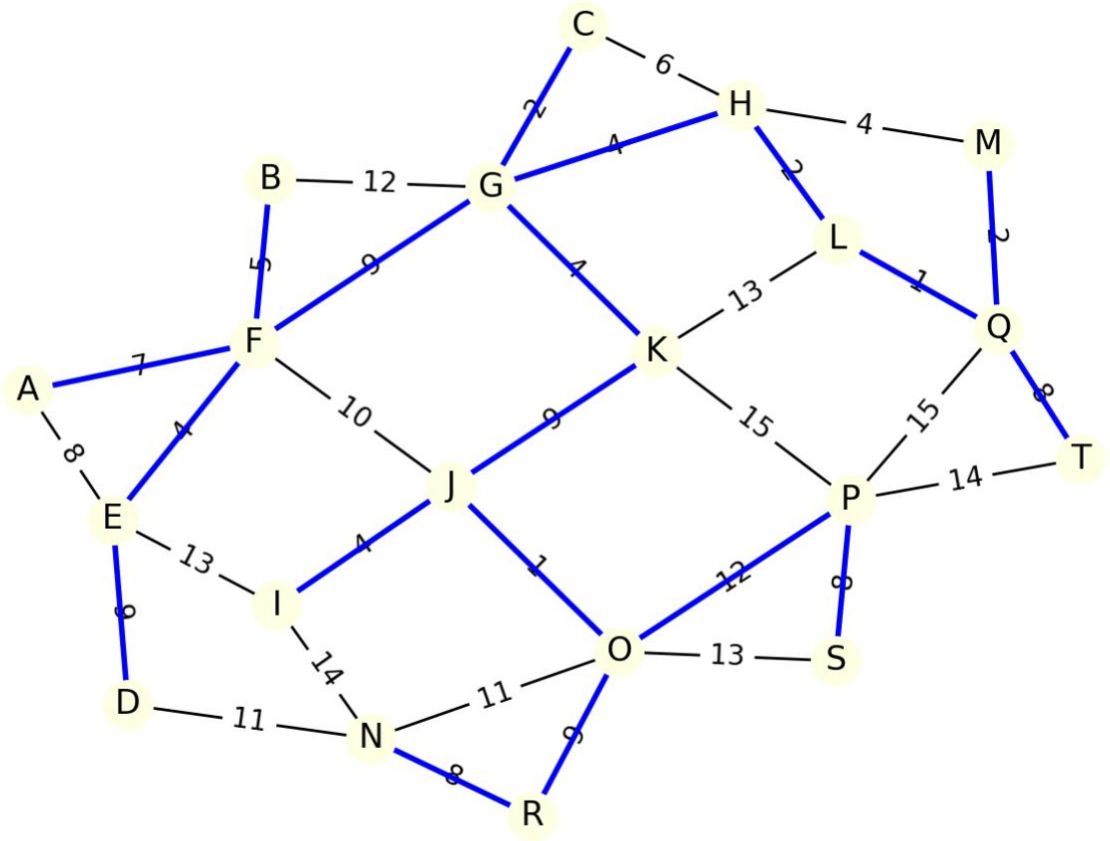
```
if __name__ == "__main__":
    # build the main NetworkX graph
    G = build_graph()
    # copy the graph into a directed graph
    DG = G.to_directed()
    # run MST algorithms of Prim and Kruskal
    P = nx.minimum_spanning_tree(G, weight="weight", algorithm="prim")
```

```python
def show_multi_graph(G, T) -> None:  3 usages
    """Shows the source graph G with an overlay of another algorithm such as MST.

    Visualizes the source G and then overlays the resulting edges of other algorithms
    such as kruskal, prim, bellman, etc...

    Parameters
    ----------
    G: NetworkX Graph
    T: NetworkX Graph
        The graph returned from other NetworkX algorithms.

    Returns
    --------
    None

    See Also
    --------
    For references of visualizing see
    https://networkx.org/documentation/stable/auto_examples/index.html
    """
    pos = nx.spring_layout(G, weight=None, seed=7)
    nx.draw(G, pos, with_labels=True, node_color="lightyellow")
    # get the weights of each edge
    edge_weights = nx.get_edge_attributes(G, name: "weight")
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_weights)
    nx.draw_networkx_edges(T, pos, edge_color="blue", width=2)
    plt.show()
```

Prim's algorithm finds the shortest path from an arbitrary root that spans all vertices in graph G eventually forming a minimum spanning tree. Each step adds to the MST a light edge that connects the tree to an isolated vertex. Essentially the tree is growing without ever being disconnected unlike Kruskal's algorithm. The light edge is also a safe edge. It is a safe edge since it won't create a cycle and violate the property of MST. During execution all vertices not in the tree reside in min-priority queue based on their key attribute. Key is the min weight of any edge connecting to a vertex.

This output is correct because we can choose any vertex to trace the path of the algorithm and see the choices made. For example, from Vertex A our first choice is going to F. From there our options have expanded to vertices B, G, J, or E. Prim's algorithm will always choose the lowest path cost first while growing out the tree since it is a greedy algorithm. The next choice is the edge (f,e) with a path cost of 4. This also keeps the MST properties such that the subset of edges does not form a cycle by only adding safe edges.

3. Kruskal's algorithm output based on input graph from number 1



Associated code

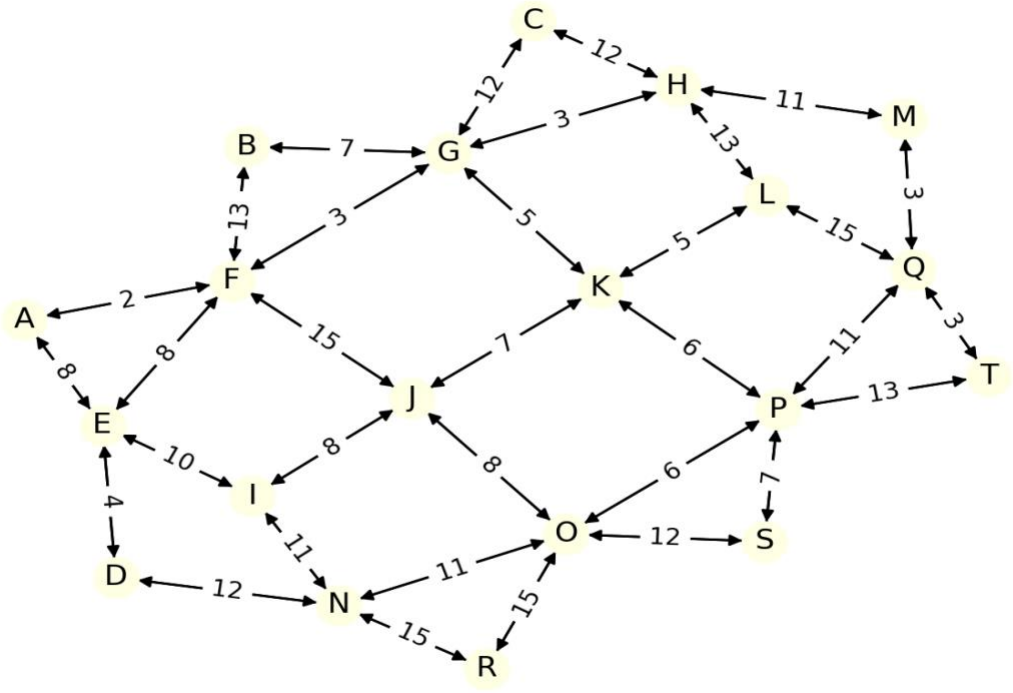
```
# run MST algorithms of Prim and Kruskal
P = nx.minimum_spanning_tree(G, weight="weight", algorithm="prim")
K = nx.minimum_spanning_tree(G, weight="weight", algorithm="kruskal")
```

```python
def show_multi_graph(G, T) -> None:  3 usages
    """Shows the source graph G with an overlay of another algorithm such as MST.

    Visualizes the source G and then overlays the resulting edges of other algorithms
    such as kruskal, prim, bellman, etc...

    Parameters
    ----------
    G: NetworkX Graph
    T: NetworkX Graph
        The graph returned from other NetworkX algorithms.

    Returns
    -------
    None

    See Also
    --------
    For references of visualizing see
    https://networkx.org/documentation/stable/auto_examples/index.html
    """
    pos = nx.spring_layout(G, weight=None, seed=7)
    nx.draw(G, pos, with_labels=True, node_color="lightyellow")
    # get the weights of each edge
    edge_weights = nx.get_edge_attributes(G, name: "weight")
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_weights)
    nx.draw_networkx_edges(T, pos, edge_color="blue", width=2)
    plt.show()
```

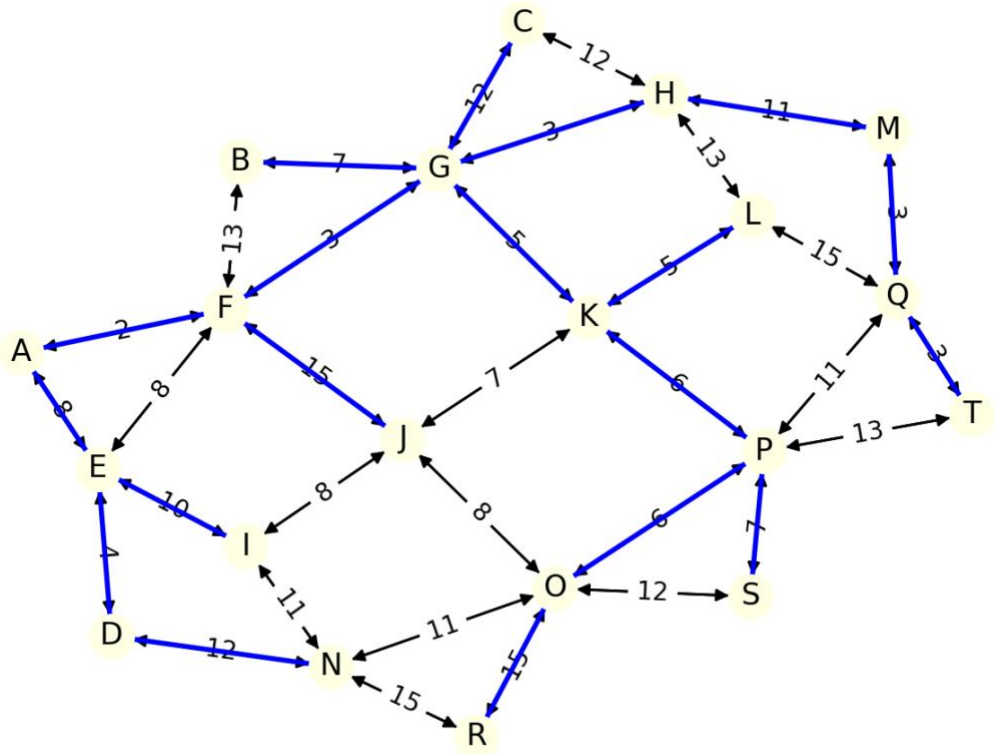
Kruskal's algorithm operates by first sorting all the edges G.E into nondecreasing order by weight. It will then iterate over the edges and add the lowest weight if they don't form a cycle. It does this by making sure the endpoints of the edges do not belong to the same tree. If they did then a cycle would form. This means Kruskal's algorithm is not a continuously growing tree like Prim's algorithm but rather a forest. Kruskal's algorithm is also greedy as it selects the local optimal choice.

The output above is correct as we sort the edges into lowest weight and have a starting point of edge (j,o) or (l,q) both with weights of 1. After selecting both we can see how a forest is forming since neither edge has connected endpoints. As we continue selecting lowest weight edges, we run into edge (h,m) with weight 4. However, at this point the edge (h,m) has vertices which are part of the same tree so this is an example of Kruskal's algorithm discarding an edge selection based on FIND-SET for vertices H and M.

4. Bellman-Ford input graph



Output Graph



Related code to Bellman–Ford

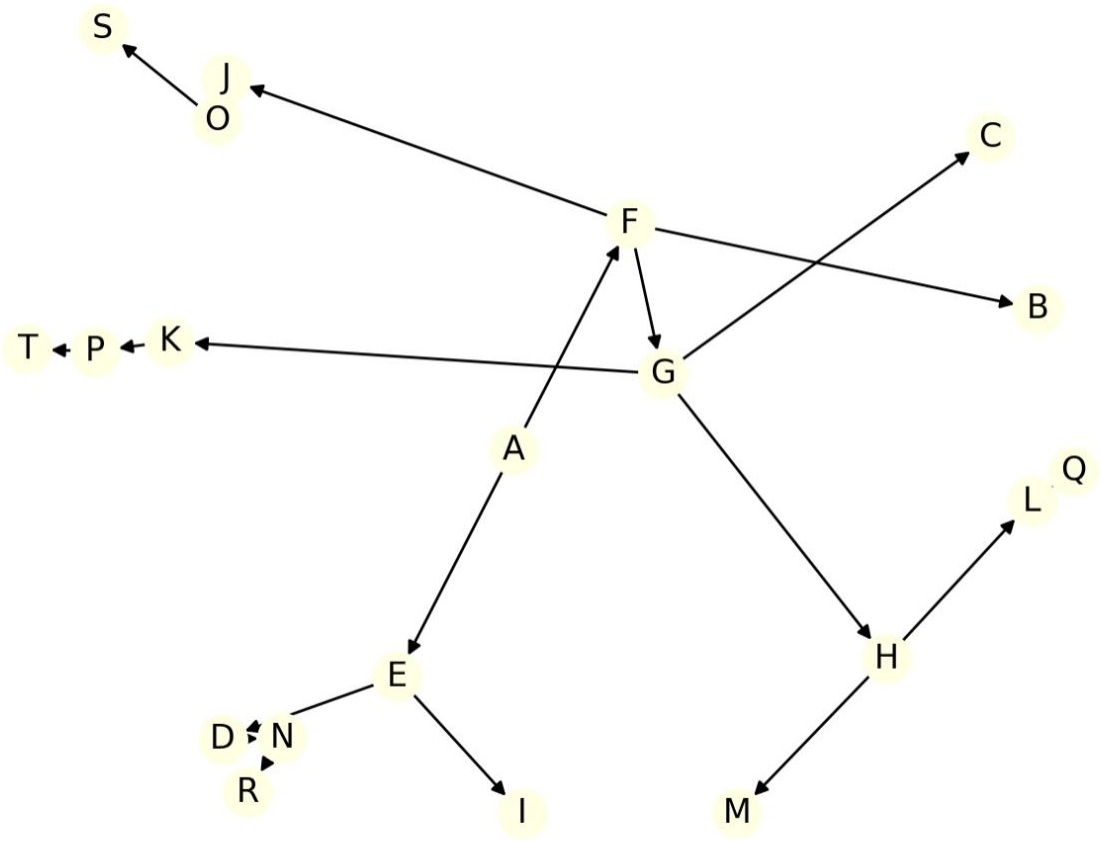
```python
def bellman(G, source_node: str) -> list[tuple[str, str]]:  1 usage
    """Runs the bellman single source algorithm on graph G.

    The single source bellman returns distance and path. Use the path
    which shows shortest distance from source_node to all other nodes to build
    the edges for visualizing the algorithm

    Parameters
    ----------
    G: NetworkX Graph
    source_node: str
        Node at which to start the Bellman-ford algorithm.

    Returns
    --------
    bellman_edges: list[tuple[str,str]]
        List of tuples that represent the edges (u,v) bellman travels
    """
    _, path = nx.single_source_bellman_ford(G, source_node)
    # k:string - node, v:list[str] - path to that node from source
    # use this to build the edges bellman travels
    bellman_edges: list[tuple[str, str]] = []
    for k, v in path.items():
        if k == source_node:
            continue
        edges = [(v[e], v[e + 1]) for e in range(len(v) - 1)]
        bellman_edges.extend(edges)

    return bellman_edges
```

Bellman-Ford works by iterating over every edge |G.V| -1 times and relaxing them. As it relaxes them it updates their distance and parent attributes by comparing the distance of vertex v (v.d) with the distance of vertex u (u.d) plus the weight (distance) from u to v. The algorithm will return FALSE if the graph contains a negative weight cycle. If there is no negative weight cycle the algorithm will produce shortest paths and their weights for all vertices reachable from the source.

The output is correct by first noticing there are no negative weight cycles and thus the algorithm did not return FALSE. Since there are no negative weight cycles the output returns the distances and paths which resulted from relaxing the edges G.V -1 times. The paths were used to generate edges for visualization. Given the source node is A we can choose any target node and follow the path as shown in blue and calculate the distance. For simplicity, let the target node be D. There are numerous possible paths from A to D however the shortest path is A -> E -> D. Furthermore, the vertices, A, E, and F exhibit the Triangle inequality property of shortest paths. Such that, shortest path of (s,v) in this case (a, f) is less than the shortest path of (a,e) + the weight of edge (e,f).

5. BFS

The input is graph is the same initial graph from number 1. BFS doesn't care about the weights so its fine to leave them. The output is a BFS Tree visualized as with the root node being A



The code for BFS

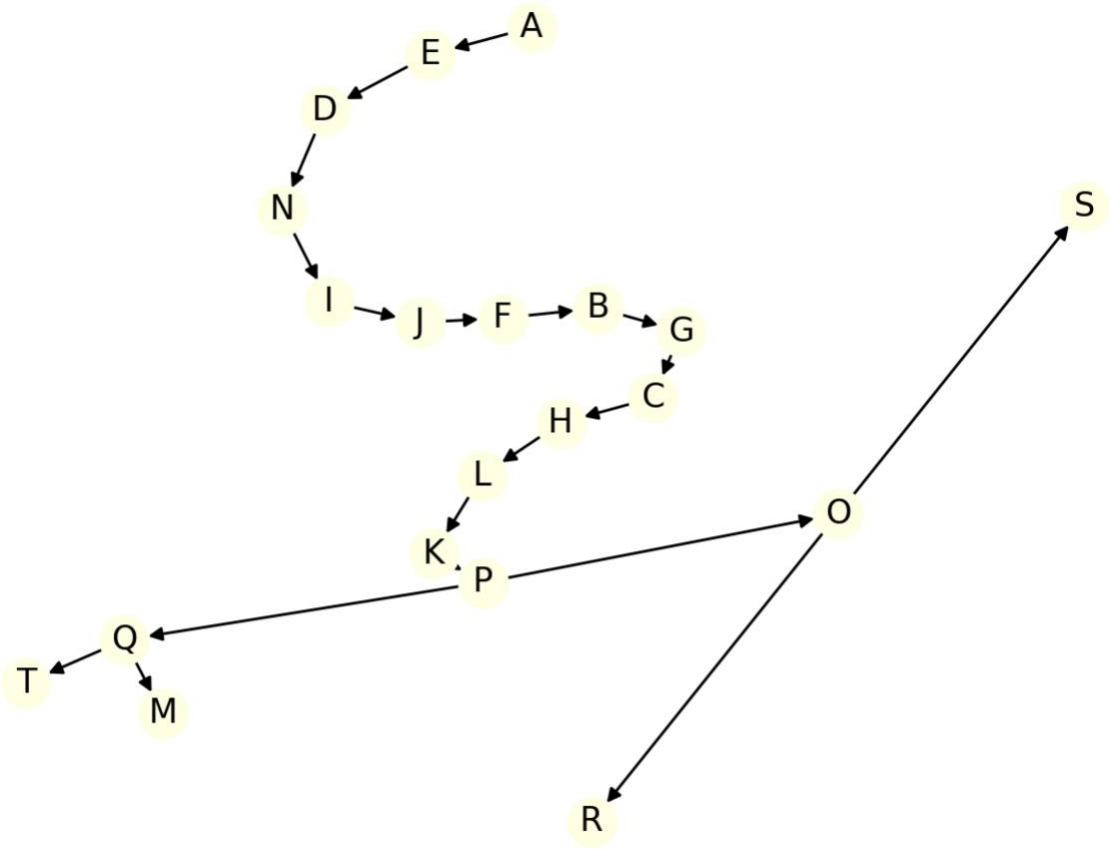
```
BFS_Tree = nx.bfs_tree(G, source: "A")
show_tree(BFS_Tree)
```

```python
def show_tree(G) -> None:  2 usages
    """Visualizes source graph G.

    Parameters
    ----------
    G: NetworkX Graph

    Returns
    --------
    None

    See Also
    --------
    For references of visualizing see
    https://networkx.org/documentation/stable/auto_examples/index.html
    """
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color="lightyellow")
    plt.show()
```

BFS expands the frontier between discovered and undiscovered vertices uniformly across the breadth meaning it discovers all vertices of its current level k before going to k + 1. It keeps track of which vertices have been discovered and explored by colors; white, gray, and black which represent undiscovered, discovered, and explored. It also keeps track of a distance property (u.d) for vertex u from source vertex s. BFS uses a FIFO queue to keep track of the order for which vertices to explore.

The result is the BFS tree which exhibits the tree property of edges (19) $|E| = |V| - 1$. We can also notice from graph G to subgraph G all vertices were reachable from the source vertex A. Also, expanded along the breadth frontier by picking a vertex like E from the from original graph and noticing all its adjacent vertices in the BFS Tree are at level k + 1 which was a result of the BFS FIFO queue.

6. DFS
   Input graph is the same as number 1. Output graph as follows



Code

```
DFS_Tree = nx.dfs_tree(G, source: "A")
show_tree(DFS_Tree)
```

Show_tree is the same function used in BFS

DFS searches deeper in the graph than BFS by exploring edges out of the most recently discovered vertex v that still has unexplored edges exiting the vertex. It can also jump around as it explores every vertex of a graph creating a depth-first forest instead of a single tree. Once it reaches a vertex with no more exiting edges it “backtracks” to its parents to explore edges exiting those vertices. Unlike BFS, DFS keeps track of a discovery time and a finish time. It marks the finish time when the adjacency list has been examined completely. The book’s algorithm utilizes recursion which is a stack instead of BFS FIFO queue.

The correctness of the output is evident by the deep single chain of nodes. This occurs because of the stack structure and process of DFS expanding deeper first instead of breadth across the frontier. Meaning if we choose a vertex E in the original graph then its vertices will no longer be at a level k + 1 in the resulting tree. Noticed by vertex I, which is adjacent to E in the parent graph, is now multiple levels below E in the DFS tree.