

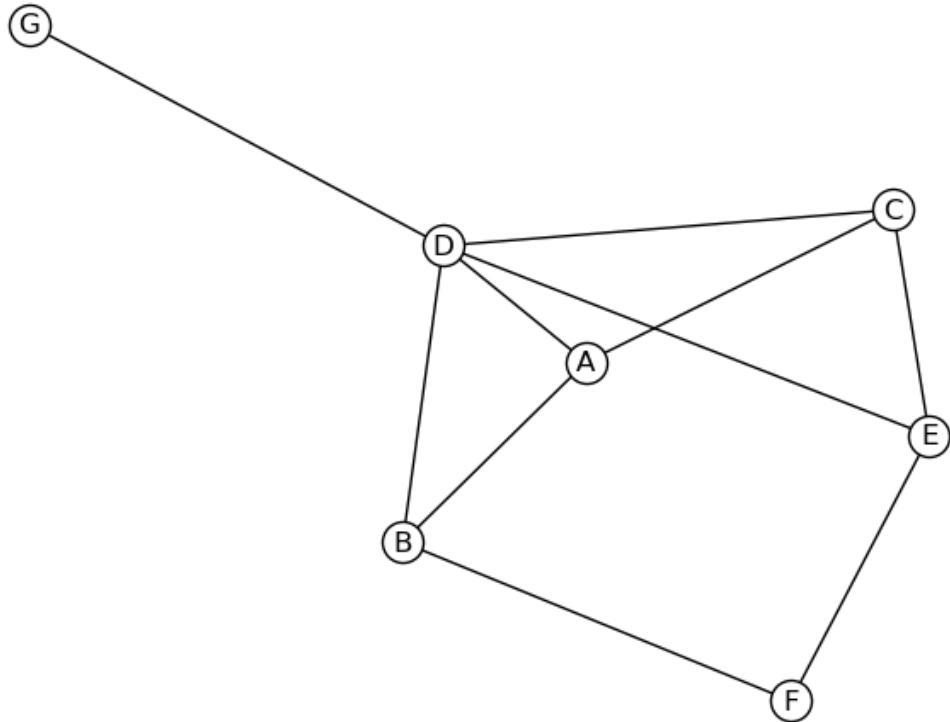
Search

September 30, 2025

```
[2]: import numpy as np
      import matplotlib.pyplot as plt
      import networkx as nx
```

```
[3]: # create a sample graph
G = nx.Graph()
edges = [
    ("A", "B"), ("A", "C"),
    ("B", "D"), ("C", "D"),
    ("C", "E"), ("D", "E"),
    ("E", "F"), ("F", "B"),
    ("D", "G"), ("D", "A")
]
G.add_edges_from(edges)

# Draw the graph
options = {
    "font_size": 12,
    "node_size": 300,
    "node_color": "white",
    "edgecolors": "black",
    "linewidths": 1,
    "width": 1,
    "with_labels": True,
}
nx.draw(G, **options)
plt.show()
```



1 Depth first search

```
[4]: # List a DFS path in preorder traversal
print("DFS order starting from A:")
print(list(nx.dfs_preorder_nodes(G, source="A")))

# List edges explored from A
print("\nList edges explored from A:")
print(list(nx.dfs_edges(G, source="A")))

# Find a single path from source to target using DFS
def dfs_path(graph, start, goal, path=None, visited=None):
    if path is None: path = [start]
    if visited is None: visited = set([start])

    if start == goal:
        return path # Found a path

    for neighbor in graph.neighbors(start):
```

```

    if neighbor not in visited:
        visited.add(neighbor)
        result = dfs_path(graph, neighbor, goal, path + [neighbor], visited)
        if result: # Stop at the first valid path
            return result
    return None

path = dfs_path(G, "A", "F")
print("\nOne DFS path from A to F:")
print(path)

# Find all DFS paths from source to target
def dfs_all_paths(graph, start, goal, path=None):
    if path is None: path = [start]
    if start == goal:
        yield path
    for neighbor in graph.neighbors(start):
        if neighbor not in path:
            yield from dfs_all_paths(graph, neighbor, goal, path + [neighbor])

print("\nAll DFS paths from A to F:")
for p in dfs_all_paths(G, "A", "F"):
    print(p)

```

DFS order starting from A:

```
['A', 'B', 'D', 'C', 'E', 'F', 'G']
```

List edges explored from A:

```
[('A', 'B'), ('B', 'D'), ('D', 'C'), ('C', 'E'), ('E', 'F'), ('D', 'G')]
```

One DFS path from A to F:

```
['A', 'B', 'D', 'C', 'E', 'F']
```

All DFS paths from A to F:

```
['A', 'B', 'D', 'C', 'E', 'F']
['A', 'B', 'D', 'E', 'F']
['A', 'B', 'F']
['A', 'C', 'D', 'B', 'F']
['A', 'C', 'D', 'E', 'F']
['A', 'C', 'E', 'D', 'B', 'F']
['A', 'C', 'E', 'F']
['A', 'D', 'B', 'F']
['A', 'D', 'C', 'E', 'F']
['A', 'D', 'E', 'F']
```

1.0.1 Breadth first search (BFS)

```
[5]: from collections import deque

def bfs_path(graph, start, goal):
    queue = deque([[start]])      # queue of paths
    visited = {start}

    while queue:
        path = queue.popleft()
        node = path[-1]

        if node == goal:
            return path

        for neighbor in graph.neighbors(node):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(path + [neighbor])

    return None

# BFS order starting from A
print("BFS order from A:")
print(list(nx.bfs_tree(G, source="A")))          # BFS tree nodes
print(list(nx.bfs_edges(G, source="A")))          # BFS tree edges

# from the manual function
path = bfs_path(G, "A", "F")
print("\nManual BFS path from A to F:")
print(path)

# Single shortest path from A to F
shortest_path = nx.shortest_path(G, source="A", target="F")
print("\nShortest path from A to F (BFS):")
print(shortest_path)

# All shortest paths (if multiple of equal length)
all_paths = list(nx.all_shortest_paths(G, source="A", target="F"))
print("\nAll shortest paths from A to F:")
for p in all_paths:
    print(p)

# All paths
from networkx.algorithms.traversal.breadth_first_search import bfs_tree

nodes = list(bfs_tree(G, source='A'))
```

```
print(nodes)
```

BFS order from A:

```
['A', 'B', 'C', 'D', 'F', 'E', 'G']
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'F'), ('C', 'E'), ('D', 'G')]
```

Manual BFS path from A to F:

```
['A', 'B', 'F']
```

Shortest path from A to F (BFS):

```
['A', 'B', 'F']
```

All shortest paths from A to F:

```
['A', 'B', 'F']
['A', 'B', 'C', 'D', 'F', 'E', 'G']
```

```
[6]: import collections
```

```
def bfs_all_shortest_paths_from_source(G, source, cutoff=None):
    # BFS to compute distances and predecessor sets at shortest distance
    dist = {source: 0}
    preds = collections.defaultdict(set)  # node -> set of predecessor nodes
    Q = collections.deque([source])

    while Q:
        u = Q.popleft()
        if cutoff is not None and dist[u] >= cutoff:
            continue
        for v in G.neighbors(u):
            # First time seen: set distance and queue
            if v not in dist:
                dist[v] = dist[u] + 1
                preds[v].add(u)
                Q.append(v)
            # If same shortest layer, record one more predecessor
            elif dist[v] == dist[u] + 1:
                preds[v].add(u)

    # backtrack (DFS-style) through precedents to enumerate all shortest paths
    # use cache paths to avoid recomputation
    cache = {source: [[source]]}

    def build_paths(t):
        if t in cache:
            return cache[t]
        all_paths = []
        for p in preds[t]:
```

```

        for path in build_paths(p):
            all_paths.append(path + [t])
        cache[t] = all_paths
    return all_paths

# build path lists only for nodes we reached
result = {}
for t in dist.keys():
    result[t] = build_paths(t)
return result

all_shortest = bfs_all_shortest_paths_from_source(G, source="A")
for t, plist in all_shortest.items():
    print(f"A --> {t}:")
    for p in plist:
        print("    ", p)

```

```

A --> A:
    ['A']
A --> B:
    ['A', 'B']
A --> C:
    ['A', 'C']
A --> D:
    ['A', 'D']
A --> F:
    ['A', 'B', 'F']
A --> E:
    ['A', 'C', 'E']
    ['A', 'D', 'E']
A --> G:
    ['A', 'D', 'G']

```

```
[7]: def bfs_all_shortest_paths_all_pairs(G, cutoff=None):
    result = {}
    for s in G.nodes():
        result[s] = bfs_all_shortest_paths_from_source(G, s, cutoff=cutoff)
    return result

# All shortest paths from a source to a target
pairs = bfs_all_shortest_paths_all_pairs(G)

print(pairs["A"]["F"])  # all shortest A --> F paths
print(pairs["A"]["D"])

```

```

[[['A', 'B', 'F']]
[['A', 'D']]]

```

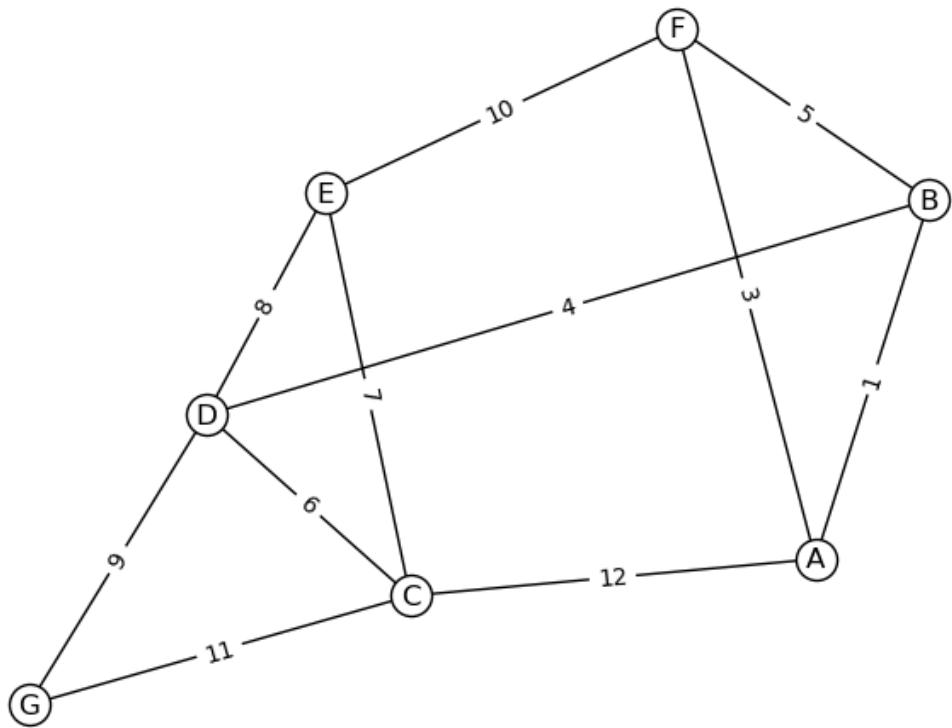
```
[8]: # weighted graph
G = nx.Graph()
edges = [
    ("A", "B", 1), ("A", "C", 12), ("A", "F", 3),
    ("B", "D", 4), ("B", "F", 5),
    ("C", "D", 6), ("C", "E", 7), ("C", "G", 11),
    ("D", "E", 8), ("D", "G", 9),
    ("E", "F", 10),      # or G.add_edge("B", "C", weight=1.7)
]
G.add_weighted_edges_from(edges)

# check all weighted edges
for u, v, d in G.edges(data=True):
    print(u, v, d["weight"])
```

A B 1
A C 12
A F 3
B D 4
B F 5
C D 6
C E 7
C G 11
F E 10
D E 8
D G 9

```
[9]: # Draw the graph
options = {
    "font_size": 12,
    "node_size": 300,
    "node_color": "white",
    "edgecolors": "black",
    "linewidths": 1,
    "width": 1,
    "with_labels": True,
}

pos = nx.spring_layout(G, seed=10)
nx.draw(G, pos, **options)
labels = nx.get_edge_attributes(G, "weight")
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.show()
```



1.0.2 Dijkstra algorithm

```
[9]: import networkx as nx
import matplotlib.pyplot as plt

# Create a weighted graph
G = nx.DiGraph()
G.add_weighted_edges_from([
    ('A', 'B', 8), ('A', 'C', 11), ('A', 'D', 9),
    ('B', 'E', 10),
    ('C', 'F', 8), ('C', 'G', 8),
    ('D', 'B', 6), ('D', 'C', 3), ('D', 'E', 1),
    ('E', 'H', 2),
    ('F', 'D', 12), ('F', 'H', 5),
    ('G', 'F', 7),
    ('H', 'G', 4)
])
source = 'A'
```

```

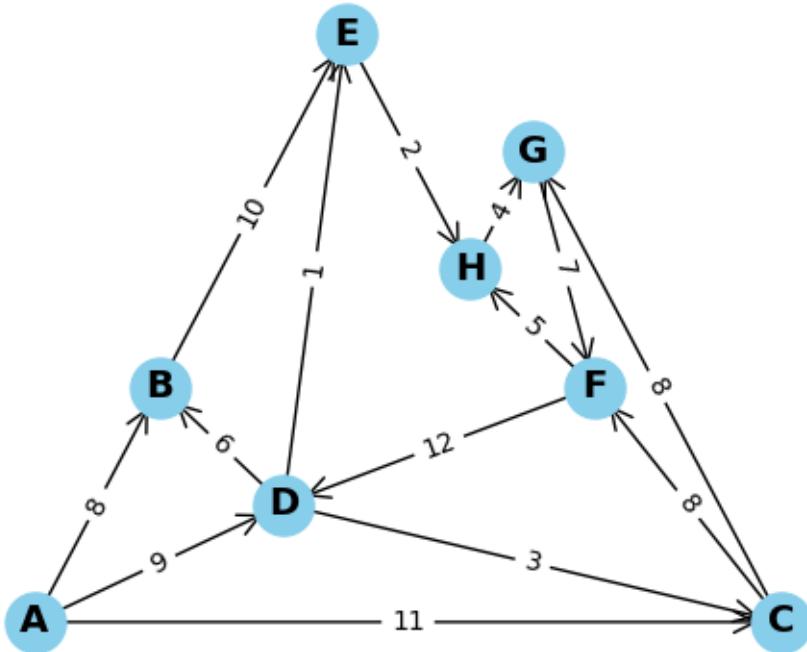
# Distances from A to all nodes
distances, paths = nx.single_source_dijkstra(G, source)
print("Distances:", distances)      # shortest distance to every node
print("Path to D:", paths['D'])    # actual path to D

pos = nx.planar_layout(G)      # positions for nodes

plt.figure(figsize=(5,4))
nx.draw_networkx_nodes(G, pos, node_size=500, node_color="skyblue")
nx.draw_networkx_edges(G, pos, arrowstyle="->", arrowsize=20)
nx.draw_networkx_labels(G, pos, font_size=14, font_weight="bold")
# Draw weights as edge labels
labels = nx.get_edge_attributes(G, "weight")
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.axis("off")
plt.tight_layout()
plt.show()

```

Distances: {'A': 0, 'B': 8, 'D': 9, 'E': 10, 'C': 11, 'H': 12, 'G': 16, 'F': 19}
Path to D: ['A', 'D']



[]: # Dijkstra algorithm for a single target for unweighted graph

```

length, path = nx.single_source_dijkstra(G, source="A", target="G", weight="weight")
print("Cost A-->G:", length, "Path:", path)
print()

# Dijkstra algorithm for all paths
print('All paths:')
for n, (dist, path) in nx.all_pairs_dijkstra(G):
    print(path)
print()

# all shortest simple paths from the single source
paths = nx.single_source_all_shortest_paths(G, source="A", weight="weight")
print('From A to all nodes:')
for s, p in paths:
    print(f'from {s} to {p}')
print()

```

Cost A-->G: 14 Path: ['A', 'B', 'D', 'G']

All paths:

```

{'A': ['A'], 'B': ['A', 'B'], 'C': ['A', 'B', 'D', 'C'], 'F': ['A', 'F'], 'D': ['A', 'B', 'D'], 'E': ['A', 'F', 'E'], 'G': ['A', 'B', 'D', 'G']}
{'B': ['B'], 'A': ['B', 'A'], 'D': ['B', 'D'], 'F': ['B', 'A', 'F'], 'C': ['B', 'D', 'C'], 'E': ['B', 'D', 'E'], 'G': ['B', 'D', 'G']}
{'C': ['C'], 'A': ['C', 'D', 'B', 'A'], 'D': ['C', 'D'], 'E': ['C', 'E'], 'G': ['C', 'G'], 'B': ['C', 'D', 'B'], 'F': ['C', 'D', 'B', 'A', 'F']}
{'F': ['F'], 'A': ['F', 'A'], 'B': ['F', 'A', 'B'], 'E': ['F', 'E'], 'C': ['F', 'A', 'B', 'D', 'C'], 'D': ['F', 'A', 'B', 'D'], 'G': ['F', 'A', 'B', 'D', 'G']}
{'D': ['D'], 'B': ['D', 'B'], 'C': ['D', 'C'], 'E': ['D', 'E'], 'G': ['D', 'G'], 'A': ['D', 'B', 'A'], 'F': ['D', 'B', 'A', 'F']}
{'E': ['E'], 'C': ['E', 'C'], 'D': ['E', 'D'], 'F': ['E', 'F'], 'A': ['E', 'F', 'A'], 'G': ['E', 'D', 'G'], 'B': ['E', 'D', 'B']}
{'G': ['G'], 'C': ['G', 'C'], 'D': ['G', 'D'], 'B': ['G', 'D', 'B'], 'E': ['G', 'D', 'E'], 'D': ['G', 'E'], 'A': ['G', 'D', 'B', 'A'], 'F': ['G', 'D', 'B', 'A', 'F']}

```

From A to all nodes:

```

from A to [['A']]
from B to [['A', 'B']]
from C to [['A', 'B', 'D', 'C']]
from F to [['A', 'F']]
from D to [['A', 'B', 'D']]
from E to [['A', 'F', 'E'], ['A', 'B', 'D', 'E']]
from G to [['A', 'B', 'D', 'G']]

```

1.0.3 TSP

```
[ ]: tsp = nx.approximation.traveling_salesman_problem  
#tsp(G, nodes = ['A', 'G']) # collection of nodes to visit  
tsp(G)
```

```
[ ]: ['A', 'F', 'E', 'C', 'D', 'G', 'D', 'B', 'A']
```