Graphs

*A **graph** is an abstract data structure consisting of a set of vertices connected by edges.*

- Trees are a specific case of a graph (acyclic, connected graphs)

Examples:
- Messaging: the traveling salesman problem, postal routes
- Communication networks
- Traffic management: flow problems, minimum congestion paths, ...
- Air navigation (aircraft in sky corridors!)
- Closed transportation system (closed circuit): goods delivery, TSP (Traveling Salesman Problem).
- Printed circuit board wiring

Definition

Graph $G = (V, E)$ with:

- V: set of nodes (vertices).
- $E \in (V \times V)$: set of edges (links) or arcs (if oriented).

Properties:

- **Connected graph**: with a path between any pair of nodes.
- **Directed graphs**: where edges have a specific direction.
- **Weighted graphs**: numerical values associated with nodes or edges.
- **Strongly connected graphs**: directed graphs where there is a path from any node to any other node.
- **Bipartite**: vertices can be divided into two sets with no edges within a set.
- **Dense graph**: with a high edge-to-vertex ratio, often with $|E| = O(|V|^2)$.
- **Path**: a sequence of connected nodes with vertice.
- **Cycle**: a path that starts and ends at the same vertex.
- **Degree**: number of edges connected to a node.

Data structures: dict

- Using a dictionnary with adjacency list (similar to trees without cycles and non-connected nodes)

```
In [3]: g = { "a" : ["d"],
              "b" : ["c"],
        "c" : ["b", "c", "d", "e"],
        "d" : ["a", "c"],
        "e" : ["c"],
        "f" : []
    }
```

```
In [4]: g.keys() # nodes
```

```
Out[4]:

dict_keys(['a', 'b', 'c', 'd', 'e', 'f'])
```

Data structures: dict

Generate the list of edges from the dict data structure.

```python
In [10]: def generate_edges(graph):
             edges = []
     for node, neighbors in graph.items():
         for neighbor in neighbors:
             edges.append((node, neighbor))
     return edges

generate_edges(g) # edges
```

Out[10]:

```
[('a', 'd'),
 ('b', 'c'),
 ('c', 'b'),
 ('c', 'c'),
 ('c', 'd'),
 ('c', 'e'),
 ('d', 'a'),
 ('d', 'c'),
 ('e', 'c')]
```

Data structures: dict

Generate the list of edges from the dict data structure (using nested list comprehension).
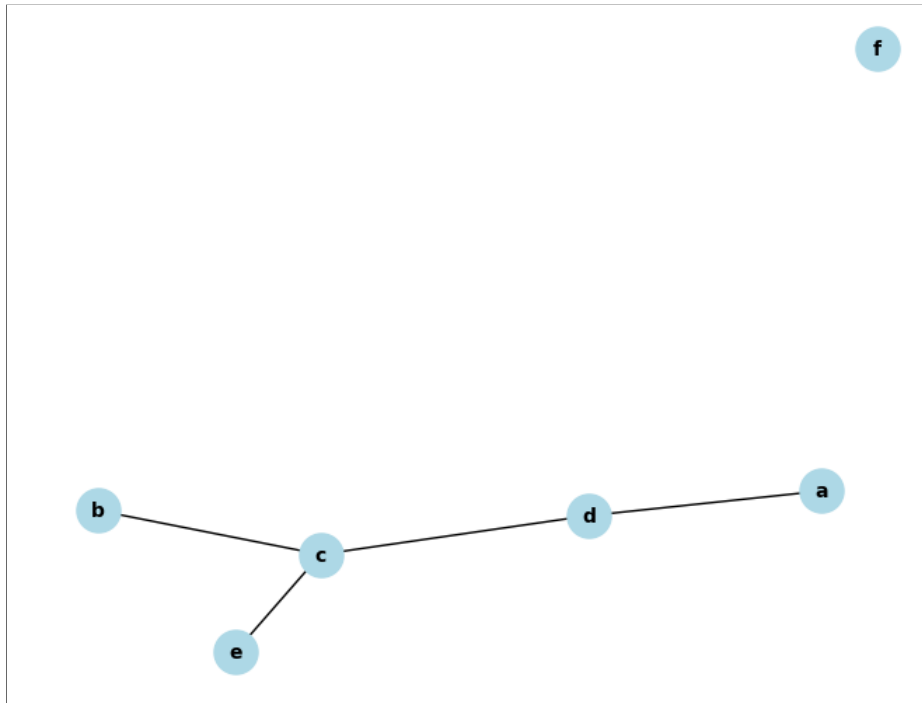
```
In [66]: [
             (vertex, neighbor)
         for vertex, neighbors in g.items()
             for neighbor in neighbors
         ]
```

Out[66]:

```
[('a', 'd'),
 ('b', 'c'),
 ('c', 'b'),
 ('c', 'c'),
 ('c', 'd'),
 ('c', 'e'),
 ('d', 'a'),
 ('d', 'c'),
 ('e', 'c')]
```

Graphs: node-link representation

```
In [67]: visualize_graph_nx(g)
```

DFS

> *Depth-First Search (DFS) starts exploring graphs at a source node, explores as far as possible along each branch before backtracking.*

- Similar than for trees
- But needs to memorize visited nodes (e.g. to prevent infinite loops)

Steps:

1. Put the source node into the stack.
2. Remove the node at the top of the stack to process it.
3. Put all unexplored neighbors into the stack (at the top).
4. If the stack is not empty, go back to step 2.

DFS with external visited list (iterative)

```
In [68]: def dfs(graph, start_node):
             visited = set()
         stack = [start_node]

         while stack:
             node = stack.pop()
             if node not in visited:
                 print(node, end=' ')
                 visited.add(node)
                 for neighbor in reversed(graph[node]):
                     if neighbor not in visited:
                         stack.append(neighbor)

dfs(g, 'a') # start from node 'a'.
```

a d c b e

DFS with external visited list (recursive)

```
In [69]: def dfs_rec(graph, start_node, visited=set()):
             if start_node not in visited:
         print(start_node, end=' ')
         visited.add(start_node)
         for neighbor in graph[start_node]:
             if neighbor not in visited:
                 dfs_rec(graph, neighbor, visited)

dfs_rec(g, 'a') # start from node 'a'.
```

a d c b e

DFS with internal visited list (recursive)

```
In [70]: def dfs(graph, start_node):
             if start_node not in graph:
         return

    print(start_node, end=' ')
    graph[start_node]['visited'] = True

    for neighbor in graph[start_node]['neighbors']:
        if not graph[neighbor]['visited']:
            dfs(graph, neighbor)

graph = {
    'A': {'neighbors': ['B', 'C'], 'visited': False},
    'B': {'neighbors': ['A', 'D', 'E'], 'visited': False},
    'C': {'neighbors': ['A', 'F'], 'visited': False},
    'D': {'neighbors': ['B'], 'visited': False},
    'E': {'neighbors': ['B', 'F'], 'visited': False},
    'F': {'neighbors': ['C', 'E'], 'visited': False}
}

dfs(graph, 'A')
```

A B D E F C

BFS

Breadth-First Search (BFS) starts exploring graphs at a source node, explores all of its neighbors at the current depth before moving on to nodes at the next depth level.

- Similar to DFS, it also requires tracking visited nodes to avoid revisiting them.

Steps for BFS:

1. Put the source node into the queue.
2. Remove the node at the front of the queue to process it.
3. Explore all unvisited neighbors of the processed node and enqueue them at the back of the queue.
4. If the queue is not empty, go back to step 2.

BFS with external visited list (iterative)

```python
In [10]: def bfs(graph, start_node):
            visited = set()
    queue = [start_node]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            for neighbor in graph.get(node, []):
                if neighbor not in visited:
                    queue.append(neighbor)

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

bfs(graph, 'A')
```

A B C D E F

BFS with backtracking

To memorize the path used to visit nodes.

```python
In [19]: def bfs_with_backtracking(graph, start_node, seeked_node):
    visited = {node: False for node in graph}
    path = {node: None for node in graph}
    queue = [start_node]
    found = False

    while queue:
        current_node = queue.pop(0)
        visited[current_node] = True

        for neighbor in graph[current_node]:
            if not visited[neighbor]:
                queue.append(neighbor)
                visited[neighbor] = True
                path[neighbor] = current_node

                if neighbor == seeked_node:
                    found = True
                    break

        if found:
            break

    if not found:
        return "Path not found"

    node = seeked_node
    path_sequence = []
    while node is not None:
        path_sequence.insert(0, node)
        node = path[node]

    return path_sequence
```

BFS with backtracking

(cont.) Path re-construction from the BFS exploration:

```python
if not found:
        return "Path not found"

    node = seeked_node
    path_sequence = []
    while node is not None:
        path_sequence.insert(0, node)
        node = path[node]

    return path_sequence
```

```
In [21]: graph = {
             'A': ['B', 'C'],
         'B': ['A', 'D', 'E'],
         'C': ['A', 'F'],
         'D': ['B'],
         'E': ['B', 'F'],
         'F': ['C', 'E']
}

start_node = 'A'
seeked_node = 'F'
path = bfs_with_backtracking(graph, start_node, seeked_node)
print(f"the path from {start_node} to {seeked_node} is {path}")
```

the path from A to F is ['A', 'C', 'F']

Graph property: path between two nodes with an edge list as input?

```
In [23]: def has_path(edges, n, start, end):
             neighbors = [[] for i in range(n)]
         for i, j in edges:
             neighbors[i].append(j)
             neighbors[j].append(i)

         stack = [start]
         visited = set(stack)
         while stack:
             cur = stack.pop()
             if cur == end:
                 return True
             for v in neighbors[cur]:
                 if v not in visited:
                     stack.append(v)
                     visited.add(v)
         return False

edges = [(0, 1), (0, 2), (1, 3), (2, 4), (3, 5), (4, 5)]
num_nodes = 6 # we set the number of unique nodes
start_node = 0; end_node = 5
result = has_path(edges, num_nodes, start_node, end_node)
print(f"the path between {start_node} and {end_node} is {result}")
```

```
the path between 0 and 5 is True
```

Find the list of unique nodes in an edge list

```python
In [24]: def unique_nodes_in_edge_list(edges):
             unique_nodes = set()
    
    for edge in edges:
        unique_nodes.add(edge[0])
        unique_nodes.add(edge[1])
    
    unique_nodes_list = sorted(list(unique_nodes))
    
    return unique_nodes_list
```

```python
In [27]: assert len(unique_nodes_in_edge_list(edges)) == 6
```

Graphs data structures: OOP

Using Oriented Object Programming.

```python
In [29]: class Graph:
             def __init__(self):
         self.graph = {}

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)

    def get_nodes(self):
        return list(self.graph.keys())

    def get_edges(self):
        edges = []
        for vertex, neighbors in self.graph.items():
            for neighbor in neighbors:
                if (vertex, neighbor) not in edges and (neighbor, vertex) not in edges:
                    edges.append((vertex, neighbor))
        return edges

    def __str__(self):
        return str(self.graph)
```

```
In [30]: g_obj = Graph()

for vertex in ["a", "b", "c", "d", "e", "f"]:
    g_obj.add_vertex(vertex)

g_obj.add_edge("a", "d")
g_obj.add_edge("b", "c")
g_obj.add_edge("c", "b")
g_obj.add_edge("c", "c")
g_obj.add_edge("c", "d")
g_obj.add_edge("c", "e")
g_obj.add_edge("d", "a")
g_obj.add_edge("d", "c")
g_obj.add_edge("e", "c")


print("Nodes:", g_obj.get_nodes())
g_obj.get_edges() == generate_edges(g) # compare with previous code
print("Edges:", g_obj.get_edges())
```

```
Nodes: ['a', 'b', 'c', 'd', 'e', 'f']
Edges: [('a', 'd'), ('b', 'c'), ('c', 'c'), ('c', 'd'), ('c', 'e')]
```

DFS using oop

(Only explores a single connex component)

```
In [33]: def dfs(self, start_vertex, visited = set()):
             stack = [start_vertex]

         while stack:
             vertex = stack.pop()
             if vertex not in visited:
                 print(vertex, end=' ')
                 visited.add(vertex)
                 neighbors = self.graph[vertex]
                 for neighbor in neighbors:
                     if neighbor not in visited:
                         stack.append(neighbor)

Graph.dfs = dfs # we update the Graph class with the dfs method
```

```
In [34]: g_obj.dfs("a")
```

a d c e b

DFS using oop

Explores all the graph components

```
In [35]: def components(self):
             visited = set()

     for vertex in self.graph:
         if vertex not in visited:
             self.dfs(vertex, visited)
             print()
Graph.components = components # update the Graph class
```

```
In [36]: g_obj.components()
```

```
a d c e b
f
```

Graph property: can a graph be n-colored?

Two adjacent vertices (connected by an edge) cannot have the same color when properly colored. Example with $n = 2$ (i.e. can a graph be colored with 2 colors).

```
In [37]: class Node:
             def __init__(self, v = None, n = []):
         self.v = v
         self.n = n
         self.visited = False

def two_color(r):

    stack = [r]

    while len(stack) > 0:
        c = stack.pop(0)
        for n in c.n:
            if(c.v == n.v): # neighbours have the same color
                return False
            if not n.visited:
                stack.append(n)
                n.visited = True

    return True
```

```
In [21]: n1 = Node("gray")
         n2 = Node("black")
n3 = Node("gray")
n4 = Node("gray")
n5 = Node("black")
n6 = Node("gray")

n1.n = [n2]
n2.n = [n1, n3, n4]
n3.n = [n2, n5]
n4.n = [n2, n5]
n5.n = [n3, n4, n6]
n6.n = [n5]

print(two_color(n1))
```

True

Data structure: Adjacency matrix

- Square: it has the same number of rows and columns.
- A 1 in a cell m_{ij} indicates a link between nodes `i` and `j`.
- A 1 on the diagonal would indicate a loop.
- It is symmetric: $m_{ij} = m_{ji}$ for an undirected graph.
- For valued graphs, cells contain values (instead of `1`).

Adjacency matrix (example)

Store in the graph G in a matrix where each row and column corresponds to a vertex in the graph (the celle value usually is set to 1 if an edge, else 0).

```
In [22]: nodes = sorted(g.keys())
         num_nodes = len(nodes)
adj_matrix = [[0] * num_nodes for _ in range(num_nodes)]
xf
for i, node in enumerate(nodes):
    connected_nodes = g[node]
    for connected_node in connected_nodes:
        j = nodes.index(connected_node)
        adj_matrix[i][j] = 1

for row in adj_matrix:
    print(row)
```

```
[0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0]
[0, 1, 1, 1, 1, 0]
[1, 0, 1, 0, 0, 0]
[0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
```

Adjacency matrix (OOP)

```
In [44]: class GraphAdj:

    def __init__(self, n):
        self.__n = n
        self.__g = [[0 for _ in range(n)] for _ in range(n)]

        for i in range(0, self.__n):
            for j in range(0, self.__n):
                self.__g[i][j] = 0


    def addEdge(self, x, y):
        if (x < 0) or (x >= self.__n):
            print("Vertex {} does not exist!".format(x))
        if (y < 0) or (y >= self.__n):
            print("Vertex {} does not exist!".format(y))

        if(x == y):
            print("Same Vertex!")
        else:
            self.__g[y][x] = 1
            self.__g[x][y] = 1

    def displayAdjacencyMatrix(self):
        for i in range(0, self.__n):
            print()
            for j in range(0, self.__n):
                print("", self.__g[i][j], end = "")

    def removeEdge(self, x, y):
        if (x < 0) or (x >= self.__n):
            print("Vertex {} does not exist!".format(x))
        if (y < 0) or (y >= self.__n):
            print("Vertex {} does not exist!".format(y))
        if(x == y):
            print("Same Vertex!")
        else:
            self.__g[y][x] = 0
            self.__g[x][y] = 0
```

```python
def displayAdjacencyMatrix(self):
    for i in range(0, self.__n):
        print()
        for j in range(0, self.__n):
            print("", self.__g[i][j], end = "")

def removeEdge(self, x, y):
    if (x < 0) or (x >= self.__n):
        print("Vertex {} does not exist!".format(x))
    if (y < 0) or (y >= self.__n):
        print("Vertex {} does not exist!".format(y))
    if(x == y):
        print("Same Vertex!")
    else:
        self.__g[y][x] = 0
        self.__g[x][y] = 0
```

```
In [45]: obj = GraphAdj(6)

obj.addEdge(0, 1)
obj.addEdge(0, 2)
obj.addEdge(0, 3)
obj.addEdge(0, 4)
obj.addEdge(1, 3)
obj.addEdge(2, 3)
obj.addEdge(2, 4)
obj.addEdge(2, 5)
obj.addEdge(3, 5)

obj.displayAdjacencyMatrix()
```

```
0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 1 1 1
1 1 1 0 0 1
1 0 1 0 0 0
0 0 1 1 0 0
```

```
In [46]: obj.removeEdge(2, 3);
         obj.displayAdjacencyMatrix();
```

```
0 1 1 1 1 0
1 0 0 1 0 0
1 0 0 0 1 1
1 1 0 0 0 1
1 0 1 0 0 0
0 0 1 1 0 0
```

Graph property: is a graph connected? (matrix)

```
In [49]: def is_connected(graph):
             n = len(graph)
         visited = [False] * n
         stack = [0] # start at node of index 0
         while stack:
             node = stack.pop()
             if not visited[node]:
                 visited[node] = True
                 for i in range(n):
                     if graph[node][i] == 1 and not visited[i]:
                         stack.append(i)
         return visited.count(True) == len(graph)

g_m = [
    [0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 1, 0, 1],
    [0, 0, 0, 1, 0]
]

is_graph_connected = is_connected(g_m) # dfs
print(f"is the graph is connected? {is_graph_connected}")
```

```
is the graph is connected? False
```

Graph property: how many connected components? (matrix)

```
In [50]: def dfs(adj_matrix, node, visited):
             visited[node] = True
     for neighbor, connected in enumerate(adj_matrix[node]):
         if connected and not visited[neighbor]:
             dfs(adj_matrix, neighbor, visited)

def count_connected_components(adj_matrix):
    num_nodes = len(adj_matrix)
    visited = [False] * num_nodes
    components = 0

    for i in range(num_nodes):
        if not visited[i]:
            dfs(adj_matrix, i, visited)
            components += 1

    return components
```

Graph property: is there a self-connected node? (matrix)

I.e is there for instance a node A → A

```
In [51]: def has_ones_in_diagonal(matrix):
             for i in range(len(matrix)):
         if matrix[i][i] == 1:
             return True
     return False
```

```
In [52]: # We check if there is a `1` in the diagonal
         has_ones_in_diagonal(g_m)
```

```
Out[52]:

False
```

Graph property: is a graph oriented? (matrix)

```
In [53]: # tip: check if the matrix is equal to its transpose
         def is_symmetric(matrix):
    rows = len(matrix)
    cols = len(matrix[0])

    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] != matrix[j][i]:
                return False
    return True
```

```
In [54]: g_empty = []
         n = 5
for i in range(n):
    row = []
    for j in range(n):
        row.append(0)
    g_empty.append(row)
```

```
In [55]: for r in g_empty:
         print(r, end="\n")
```

```
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
```

```
In [32]: is_symmetric(g_empty)
```

```
Out[32]:
True
```

Graph property: is a graph connected? (dict)

We check if the dfs equals the number of nodes.

```python
In [33]: def is_connected(graph):
             if not graph:
         return True

     visited = set()
     start_node = list(graph.keys())[0]

     def dfs(node):
         visited.add(node)
         for neighbor in graph[node]:
             if neighbor not in visited:
                 dfs(neighbor)

     dfs(start_node)

     return len(visited) == len(graph)
```

```python
In [34]: is_connected(g)
```

Out[34]:

False

Weighted graphs

Graph with numerical values associated with nodes or edges.

```
In [54]: graph_w = {
            "a": [("d", 1)],
    "b": [("c", 3)],
    "c": [("a", 1), ("b", 3), ("d", 1), ("e", 1)],
    "d": [("a", 1), ("c", 1)],
    "e": [("c", 1)],
    "f": []
}
```

Weighted graphs

```
In [49]: visualize_graph_w(graph_w)
```
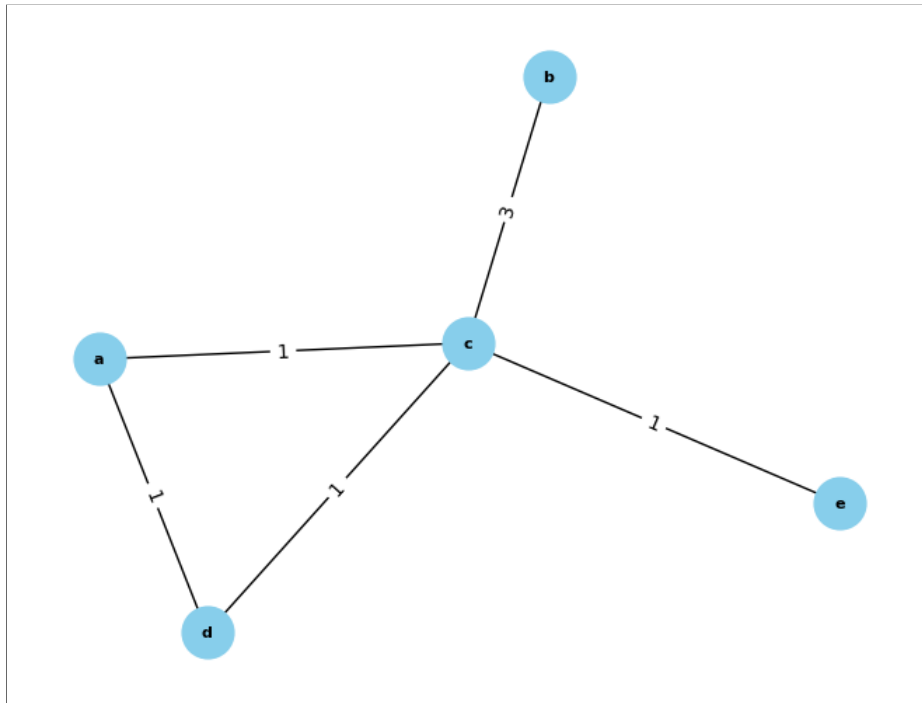
Spanning Trees

> A **Minimum Spanning Tree (MST)** of a graph is a subset of edges that connects all vertices while minimizing the total sum of the edge values.

- If a graph has N vertices, its MST (Minimum Spanning Tree) will have $N - 1$ edges.
- A graph can have multiple spanning trees, but the MST is the one with the lowest weight.
- A tree has only one spanning tree: itself.

Question: What is the minimum spanning tree of this graph?
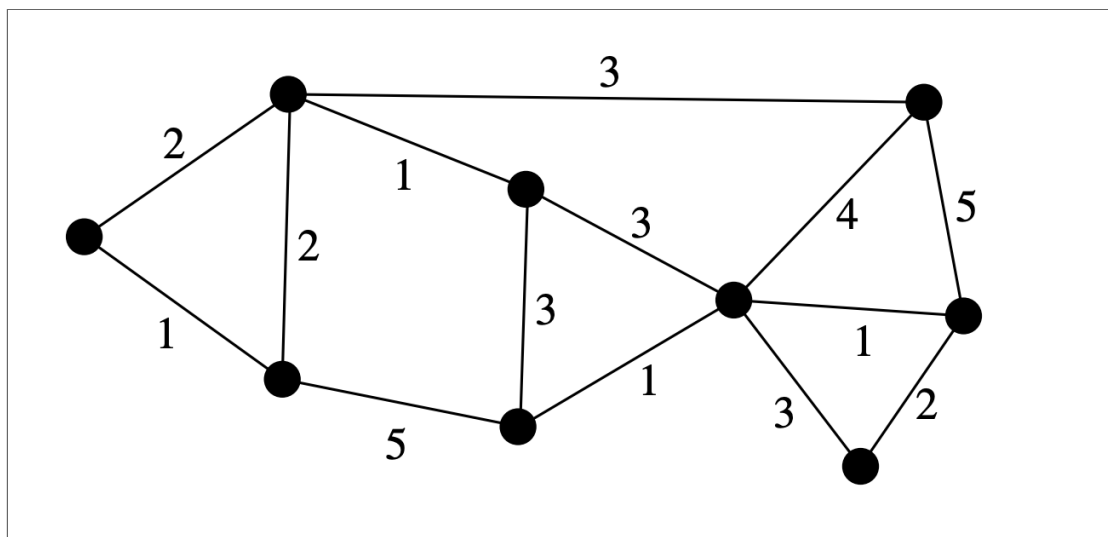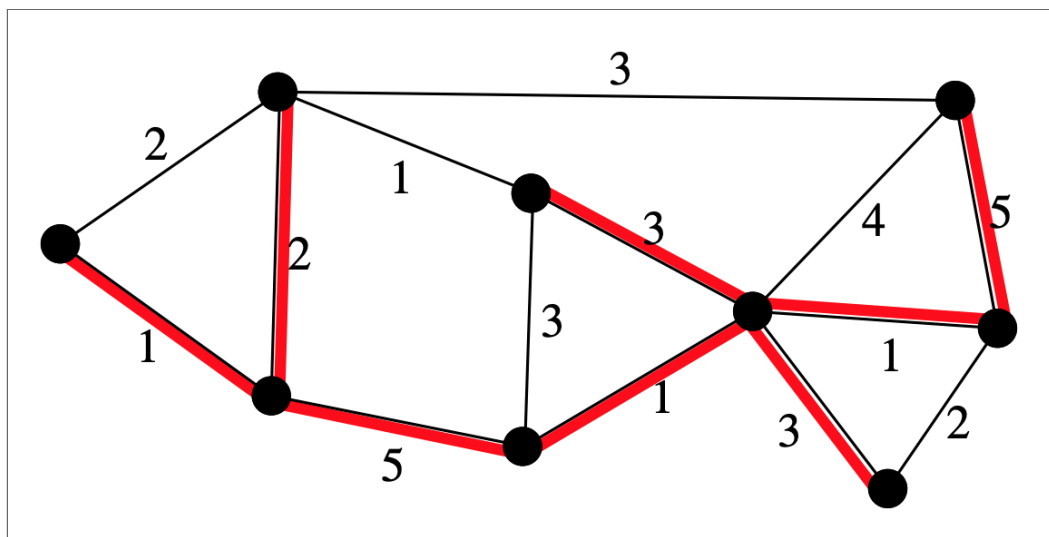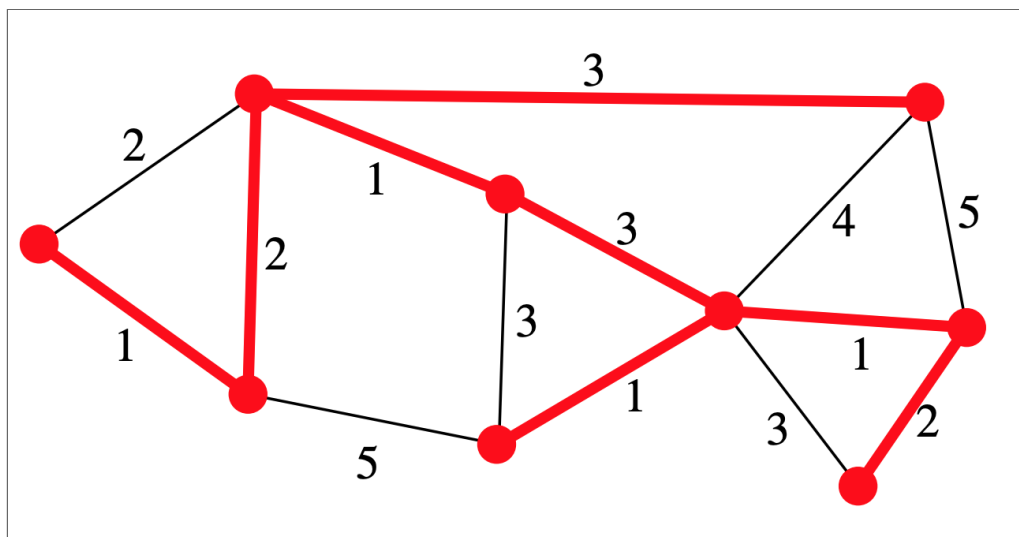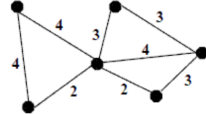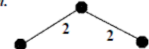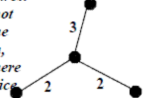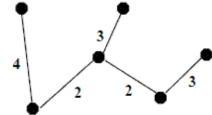
Weighted Graph MST finding: Prim's Algorithm

1. Start with an initial tree reduced to a single vertex of the graph.
2. At each iteration, expand the tree by adding the available free vertex with the smallest possible weight.
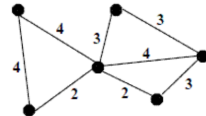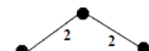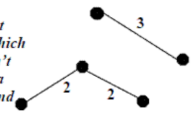3. Stop when the tree becomes spanning.
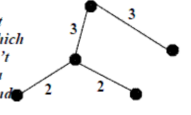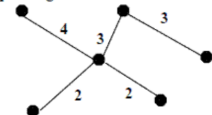
Programming Strategy?

Greedy

Prim's Algorithm

1	2	3
Given a network...........	*Choose a vertex*	*Choose the shortest edge from this vertex.*
4	5	6
Choose the nearest vertex not yet in the solution.	*Choose the next nearest vertex not yet in the solution, when there is a choice choose either.*	*Repeat until you have a minimal spanning tree.*

Kruskal's Algorithm

1	2	3
Given a network...........	*Choose the shortest edge (if there is more than one, choose any of the shortest)........*	*Choose the next shortest edge and add it.........*
4	5	6
Choose the next shortest edge which wouldn't create a cycle and add it.	*Choose the next shortest edge which wouldn't create a cycle and add it.*	*Repeat until you have a minimal spanning tree.*

Weighted Graph MST finding: Prim's Algorithm

```python
In [37]: from heapq import heapify, heappop, heappush

def prim(graph):
    mst = []
    start_vertex = list(graph.keys())[0]
    priority_queue = [(0, start_vertex)]
    visited = set()

    while priority_queue:
        weight, current_vertex = heappop(priority_queue)
        if current_vertex not in visited:
            mst.append((current_vertex, weight))
            visited.add(current_vertex)
            for neighbor, edge_weight in graph[current_vertex]:
                if neighbor not in visited:
                    heappush(priority_queue, (edge_weight, neighbor))
    return mst
```
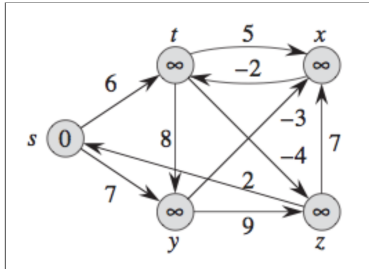
```python
In [38]: prim(graph_w)
```

```
Out[38]:

[('a', 0), ('d', 1), ('c', 1), ('b', 1), ('e', 1)]
```

Graphs: shortest paths

What is the shortest path from $s \to z$?

Approaches:

1. **BFS with local minimum (greedy):**

2. **BFS with global minimum (dynamic programming):**

3. Other?

Graphs: shortest paths (BFS)

```
In [1]: graph_s = {
            "s": [("t", 6), ("y", 7)],
        "t": [("x", 5), ("y", 8), ("z", -4)],
        "y": [("x", -3), ("z", 9)],
        "x": [("t", -2)],
        "z": [("s", 2), ("x", 7)]
    }
```

Graphs: shortest paths (Bellman-Ford)

- **Objective**: Determine the shortest paths from a single source to all other nodes in the graph.
- **Initialization**: Assign an initial distance value of 0 to the source node and infinity to all other nodes.
- **Iterative edges relaxation**:
  - Perform $|V| - 1$ iterations (V being the number of vertices).
  - For each edge (u, v), update the distance if the distance to node v through node u is shorter than the current distance to v.
- **Detection of negative cycles**:
  - After the $|V| - 1$ iterations, check for negative cycles by iterating through all edges.
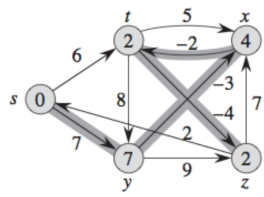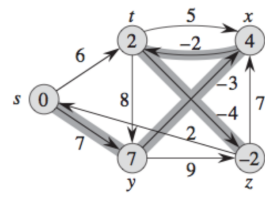 - If a shorter path is found, a negative cycle exists.
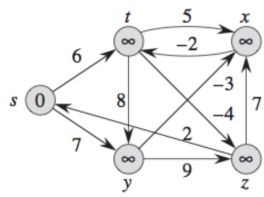
(a)
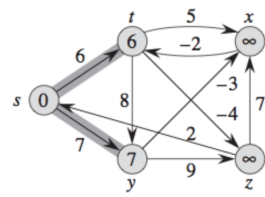
(b)
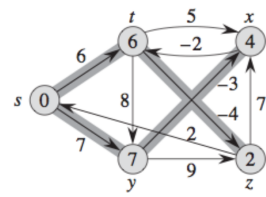
(c)

(d)

(e)

Algorithme 1 Bellman-Ford(G, w, s)

1: **pour tout** sommet $v \in V$ **faire** // Initialisation
2:     $d[v] \leftarrow \infty$, $\pi[v] \leftarrow$ NIL
3: $d[s] \leftarrow 0$
4: **pour** i de 1 à $|V| - 1$ **faire**
5:     **pour tout** arc $(u, v) \in E$ **faire** // relâchement de l'arc (u, v)
6:         **si** $d[v] > d[u] + w(u, v)$ **alors**
7:             $d[v] \leftarrow d[u] + w(u, v)$, $\pi[v] \leftarrow u$
8: **pour tout** arc $(u, v) \in E$ **faire** // détection des circuits négatifs
9:     **si** $d[v] > d[u] + w(u, v)$ **alors**
10:         **retourner Faux**
11: **retourner Vrai**

```
In [2]: def bellman_ford(graph, src):
            dist = {node: float("inf") for node in graph}
        dist[src] = 0

        for _ in range(len(graph) - 1):
            for u in graph:
                for v, w in graph[u]:
                    if dist[u] != float("inf") and dist[u] + w < dist[v]:
                        dist[v] = dist[u] + w

        for u in graph:
            for v, w in graph[u]:
                if dist[u] != float("inf") and dist[u] + w < dist[v]:
                    print("Le graphe contient des cycles négatifs")
                    return

        return dist
```

```
In [3]: bellman_ford(graph_s, 's')
```

```
Out[3]:
{'s': 0, 't': 2, 'y': 7, 'x': 4, 'z': -2}
```

Dijkstra's Algorithm

- **Objective:** Determine the shortest paths between sources S and nodes in the graph accessible from S.
- Incremental and greedy construction of a set of visited nodes E accessible from initial vertex S.
- **Initialization:** E_0 is an empty list and $G = \{S\}$.
- Move to the next step:
    - $E_{i+1} = E_i \cup \{$ node from G outside of E_i closest to S by following a path that only passes through nodes in $E_i\}$.
- The vertices entering E in ascending order of distance to S.

Warning: assumes costs > 0.

```
In [44]: graph_d = {
             "s": [("t", 6), ("y", 4)],
         "t": [("x", 3), ("y", 2)],
         "y": [("t", 1), ("x", 9), ("z", 3)],
         "x": [("z", 4)],
         "z": [("s", 7), ("x", 5)]
}
```

Algorithme 7: Algorithme de Dijkstra

Données : Un graphe orienté pondéré $G = (X, A, W)$ et un sommet $s \in X$
Résultat : Le plus court chemin de s vers tous les autres sommets de G
// V : Tableau stockant les étiquettes des sommets de G

1 Initialiser V à $+\infty$
2 $V[s] = 0$
   // P : Tableau permettant de retrouver la composition des chemins
3 Initialiser P à 0
4 $P[s] = s$
5 répéter
      // Recherche du sommet x non fixé de plus petite étiquette
6      $V_{min} = +\infty$
7   pour y *allant de 1 à N* faire
8   si y *non marqué et* $V[y] < V_{min}$ alors
9              $x \leftarrow y$
10             $V_{min} \leftarrow V[y]$

      // Mise à jour des successeurs non fixés de x
11   si $V_{min} < +\infty$ alors
12   Marquer x
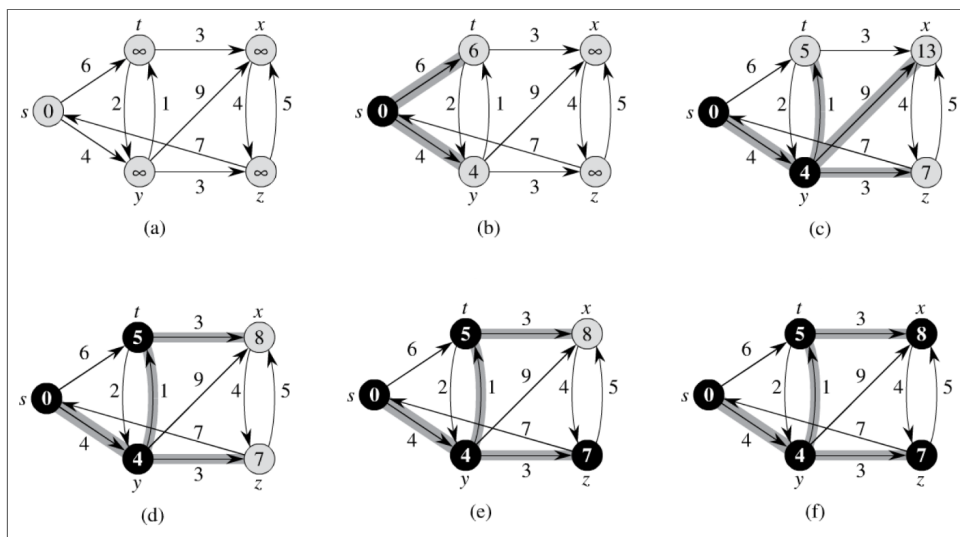13   pour *tout successeur y de x* faire
14   si y *non marqué et* $V[x] + W[x, y] < V[y]$ alors
15                 $V[y] = V[x] + W[x, y]$
16                 $P[y] = x$

17 jusqu'à $V_{min} = +\infty$

(a)   (b)   (c)

(d)   (e)   (f)

```python
In [45]: def dijkstra(graph, initial):
             visited = {initial: 0}
     path = {}
     nodes = set(graph.keys())
     while nodes:
         min_node = None
         for node in nodes:
             if node in visited:
                 if min_node is None:
                     min_node = node
                 elif visited[node] < visited[min_node]:
                     min_node = node

         if min_node is None:
             break

         nodes.remove(min_node)
         current_weight = visited[min_node]
         for edge, weight in graph[min_node]:
             weight = current_weight + weight
             if edge not in visited or weight < visited[edge]:
                 visited[edge] = weight
                 path[edge] = min_node

     return visited, path
```

```
In [46]: dijkstra(graph_d, 's')
```

Out[46]:

```
({'s': 0, 't': 5, 'y': 4, 'x': 8, 'z': 7},
 {'t': 'y', 'y': 's', 'x': 't', 'z': 'y'})
```

Summary of shortest path finding

- Principle of minimizing a cost (optimal sub-problem)
- Principle of algorithms (Bellman–Ford, Dijkstra, Floyd–Warshall) is to overestimate the weights of the vertices and adjust the cost using a *relaxation* method.
- The Bellman–Ford algorithm is similar to Dijkstra's. We find the notion of relaxation:

  $d(j) \rightarrow \min(d(j), d(x) + G(x, j)).$

- Dijkstra does not tolerate negative costs and uses a priority queue to process edges in the correct order and relax each edge only once.
- Bellman–Ford processes edges in an arbitrary order. It tolerates negative costs. For these reasons, multiple iterations might be necessary.
- Dijkstra with a cost graph of 1 resembles breadth-first search (the queue becomes a stack).

Heuristics (e.g. A*)

A* algorithm combines:

- `cost` function of reaching a destination node from the current node
- `heuristic` function estimates the cost from a node to the target node

<u>Online demo</u>

```
In [72]: import heapq

# 2D grid where walls are 'X'
grid = [
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', 'X', ' ', ' ', ' ', 'X', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', 'X', 'X', ' ', ' ', ' '],
    [' ', 'X', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    [' ', ' ', ' ', ' ', 'X', ' ', ' ', 'X', ' '],
    [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
]

# movements: up, down, left, right
movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]

def heuristic(node, goal): # Manhattan distance
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])
```

```python
In [74]: def astar(start, goal):
             open_list = []
         visited = set()
         heapq.heappush(open_list, (0, start))  # (f, node)

         while open_list:
             _, current = heapq.heappop(open_list)

             if current == goal:
                 return True  # we found a path

             visited.add(current)

             for move in movements:
                 neighbor = (current[0] + move[0], current[1] + move[1])

                 if (0 <= neighbor[0] < len(grid) and
                         0 <= neighbor[1] < len(grid[0]) and
                         grid[neighbor[0]][neighbor[1]] != 'X' and
                         neighbor not in visited):

                     f = heuristic(neighbor, goal)
                     heapq.heappush(open_list, (f, neighbor))

         return False

start_node = (0, 0)
goal_node = (5, 6)
if astar(start_node, goal_node):
    print("Path found from", start_node, "to", goal_node)
else:
    print("No path found from", start_node, "to", goal_node)
```

Path found from (0, 0) to (5, 6)

```python
In [ ]:
```