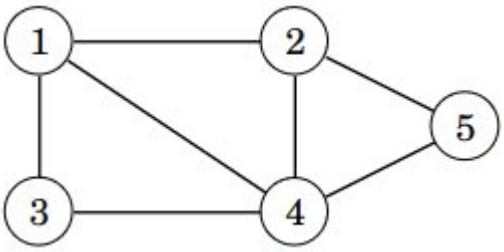
A *graph* is a data structure that consists of *nodes* and *edges*. Each edge connects two nodes.

For example, the following graph has five nodes and seven edges:



A *neighbor* of node is another node connected to it by an edge. In the example, the neighbors of the node 1 are the nodes 2, 3 and 4. The *degree* of a node is the number of its neighbors. For example, the degree of the node 1 is 3, because it has 3 neighbors.

A *path* between two nodes is a route along the edges of the graph. Here are some of the possible paths from the node 1 to the node 5:

- $1 \to 2 \to 5$
- $1 \to 4 \to 5$
- $1 \to 3 \to 4 \to 5$
- $1 \to 3 \to 4 \to 2 \to 5$

Examples of applications of graphs:

- Road network: The nodes are cities and the edges are roads. A path between two nodes is a route between two cities.
- Contact network: The nodes are people and the edges are contacts. A path between two nodes describes how people know each other.
- Communication network: The nodes are computers and the edges are connections. A path between two nodes describes how data can be transmitted.

Programming with graphs

A common way to represent a graph in programming is to have an *adjacency list* for each node. The adjacency list of a node x contains all nodes connected to x by an edge.

In Python, we can store a graph using a dictionary, where the keys are nodes and the values are adjacency lists. The example graph can be stored as follows:

```
graph = {
    1: [2, 3, 4],
    2: [1, 4, 5],
    3: [1, 4],
    4: [1, 2, 3, 5],
    5: [2, 4]
}
```

It is often useful to define a class for graphs with a method for adding edges. The class can be implemented as follows:

```
class Graph:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
        self.graph[b].append(a)
```

Here the constructor gets a list `nodes` as a parameter containing the nodes of the graph. It then creates a dictionary `graph` that stores the adjacency lists. Initially, all adjacency lists are empty, and the method `add_edge` can be used for adding an edge between the nodes `a` and `b`.

Now the example graph can be created as follows:

```
g = Graph([1, 2, 3, 4, 5])

g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 4)
g.add_edge(2, 5)
```

```
g.add_edge(3, 4)
g.add_edge(4, 5)
```

Depth-first search

Depth-first search or *DFS* is a technique for iterating through all nodes of a graph that can be reached from a given initial node by following edges. Depth-first search can be implemented using recursion given the adjacency lists.

The following code implements depth-first search:

```
class DFS:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
        self.graph[b].append(a)

    def visit(self, node):
        if node in self.visited:
            return
        self.visited.add(node)

        for next_node in self.graph[node]:
            self.visit(next_node)

    def search(self, start_node):
        self.visited = set()
        self.visit(start_node)
        return self.visited
```

The method `search` performs a depth-first search starting from the node `start_node` and returns the nodes found. The method first creates a set `visited` that contains the nodes found during the search so far. Then the method calls the recursive method `visit` that performs the search.

The method `visit` is given a node to be visited as a parameter. If the node has already been visited, the method exits. Otherwise, the node is added to the set `visited`, and then the method goes through the adjacency list of the node and

calls itself for each of the neighbors of the node.

The following code illutrates the operation of a depth-first search:

```
d = DFS([1, 2, 3, 4, 5])

d.add_edge(1, 2)
d.add_edge(1, 3)
d.add_edge(1, 4)
d.add_edge(2, 4)
d.add_edge(2, 5)
d.add_edge(3, 4)
d.add_edge(4, 5)

print(d.search(1))
```

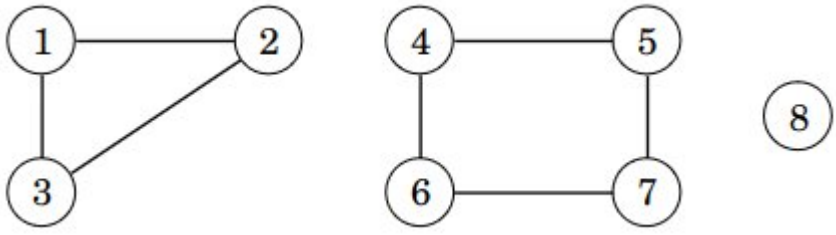
The code prints out:

```
{1, 2, 3, 4, 5}
```

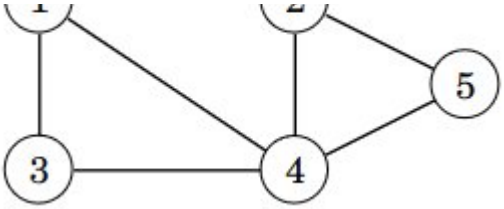
This means that the nodes 1, 2, 3, 4, and 5 were visited during the depth-first first starting from the node 1, in other words, all nodes of the graph are reachable from the node 1.

Components and connectivity

A *component* of a graph contains nodes that are reachable from each other using the edges of the graph. For example, the following graph has three components: $\{1, 2, 3\}$, $\{4, 5, 6, 7\}$ ja $\{8\}$.



A graph is *connected* if it has exactly one component, i.e., if there is a path between any two nodes. For example, the following graph is connected, since its only component is $\{1, 2, 3, 4, 5\}$.



For example in a road network, if two cities are in the same component, one can travel between the cities using roads. If a road network is connected, any city can be reached from any other city using roads.

Using depth-first search we can compute the components of a graph by iterating through all nodes and starting a new search whenever we encounter a node that is not yet in any component. The following class `Components` implements this search:

```
class Components:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
        self.graph[b].append(a)

    def visit(self, node):
        if node in self.components:
            return
        self.components[node] = self.counter

        for next_node in self.graph[node]:
            self.visit(next_node)

    def find_components(self):
        self.counter = 0
        self.components = {}

        for node in self.nodes:
            if node not in self.components:
                self.counter += 1
                self.visit(node)

        return self.components
```

The variable `counter` stores the number of components. Whenever we encounter a node that does not belong to any of the existing components, the variable is incremented by one, which correspond to creating a new component. The new component is initially empty and is then filled using a depth-first search. The dictionary `components` stores the component of each node visited so far. The current value of the variable `counter` is used as the identifier of the component during the depth-first search.

The class can be used as follows:

```
c = Components([1, 2, 3, 4, 5])

c.add_edge(1, 2)
c.add_edge(3, 4)
c.add_edge(3, 5)
c.add_edge(4, 5)

print(c.find_components()) # {1: 1, 2: 1, 3: 2, 4: 2, 5: 2}
```

In this case, the graph has two components. The first component is $\{1, 2\}$ and the second component is $\{3, 4, 5\}$.

Breadth-first search

Breadth-first search or *BFS* is another technique for iterating through the nodes a graph. Similarly to depth-first search, breadth-first search starts at a given node and visits all nodes that are reachable from the start node using edges of the graph. The difference between the two techniques is the order in which the nodes are visited.

Breadth-first search can be implemented as follows:

```
class BFS:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
        self.graph[b].append(a)
```

```
    def search(self, start_node):
        visited = set()

        queue = [start_node]
        visited.add(start_node)

        for node in queue:
            for next_node in self.graph[node]:
                if next_node not in visited:
                    queue.append(next_node)
                    visited.add(next_node)

        return visited
```

The idea is to create a list `queue` that contains the nodes to be processed. Initially, the list `queue` contains the start node. In each step of the main loop, the search takes the next node from the queue and goes through the adjacency list of the node. Any node on the adjacency list that has not been visited yet is added to the queue.

The following code illustrates depth-first search:

```
b = BFS([1, 2, 3, 4, 5])

b.add_edge(1, 2)
b.add_edge(1, 3)
b.add_edge(1, 4)
b.add_edge(2, 4)
b.add_edge(2, 5)
b.add_edge(3, 4)
b.add_edge(4, 5)

print(b.search(1))
```

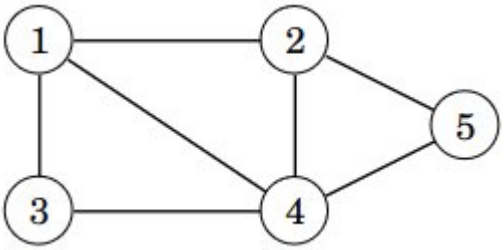
The code prints out:

```
{1, 2, 3, 4, 5}
```

Shortest paths and distances

The *shortest path* between two nodes in a graph is a path connecting the two nodes with the smallest number of edges. The *distance* of two nodes is the length of the shortest path between them.

For example, in the following graph the shortest path from the node 1 to the node 5 goes from the node 1 to the node 4 and then from the node 4 to the node 5. Since the shortest path has 2 edges, the distance from the node 1 to the node 5 is 2.



A feature of depth-first search is that it visits nodes in the order of their distance from the start node. Thus we can use depth-first search to determine the distance of each node to the start node, and to find a shortest path between two nodes.

Let us consider computing distances first. The following class computes the distance from the start node to all nodes of the graph:

```
class Distances:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
        self.graph[b].append(a)

    def find_distances(self, start_node):
        distances = {}

        queue = [start_node]
        distances[start_node] = 0

        for node in queue:
            distance = distances[node]
            for next_node in self.graph[node]:
                if next_node not in distances:
                    queue.append(next_node)
                    distances[next_node] = distance + 1
```

```
        return distances
```

The code is otherwise the same as the earlier depth-first search, but now the set `visited` has been replaced by the dictionary `distances` that stores the discovered distances. When we encounter a node whose distance is not known yet, its distance becomes one bigger than the distance of the node through which it was reached.

The class can be used as follows:

```
d = Distances([1, 2, 3, 4, 5])

d.add_edge(1, 2)
d.add_edge(1, 3)
d.add_edge(1, 4)
d.add_edge(2, 4)
d.add_edge(2, 5)
d.add_edge(3, 4)
d.add_edge(4, 5)

print(d.find_distances(1)) # {1: 0, 2: 1, 3: 1, 4: 1, 5: 2}
```

Here we computed the distance from the node 1 to all nodes of the graph:

- The distance to the node 1 is 0.
- The distance to the node 2 is 1.
- The distance to the node 3 is 1.
- The distance to the node 4 is 1.
- The distance to the node 5 is 2.

Using depth-first search we can also implement a class that finds a shortest path between two nodes:

```
class ShortestPaths:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
        self.graph[b].append(a)
```

```
    def find_path(self, start_node, end_node):
        distances = {}
        previous = {}

        queue = [start_node]
        distances[start_node] = 0
        previous[start_node] = None

        for node in queue:
            distance = distances[node]
            for next_node in self.graph[node]:
                if next_node not in distances:
                    queue.append(next_node)
                    distances[next_node] = distance + 1
                    previous[next_node] = node

        if end_node not in distances:
            return None

        node = end_node
        path = []
        while node:
            path.append(node)
            node = previous[node]

        path.reverse()
        return path
```

The method `find_path` gets the nodes `start_node` and `end_node` as parameters and finds the shortest path between these nodes.

The dictionary `distances` stores distances to the start node as before. In addition, the dictionary `previous` stores for each node the preceding node on the shortest path, i.e., the node through which it was first reached. Using the dictionary `previous`, we can trace back the shortest path from the end node back to the start node. The path is then reversed before returning it.

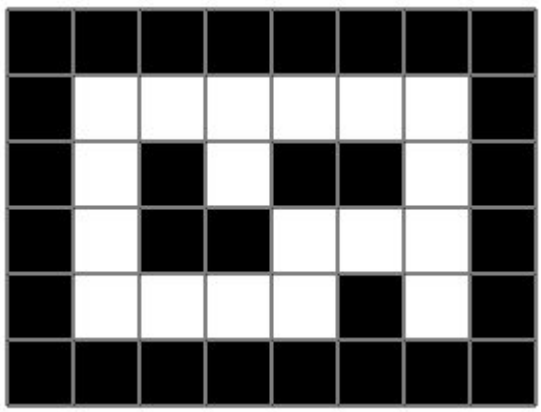
The following code illustrates the use of the class:

```
s = ShortestPaths([1, 2, 3, 4, 5])

s.add_edge(1, 2)
```

```
s.add_edge(1, 3)
s.add_edge(1, 4)
s.add_edge(2, 4)
s.add_edge(2, 5)
s.add_edge(3, 4)
s.add_edge(4, 5)

print(s.find_path(2, 4)) # [2, 4]
print(s.find_path(1, 5)) # [1, 2, 5]
print(s.find_path(5, 1)) # [5, 2, 1]
```

Labyrinth as a graph

Consider the following labyrinth, where the white squares are floor and the black squares are wall:



We can use depth- or breadth-first search to find routes in the labyrinth. We can use graph nodes to represent the floor squares and edges to tell which squares are adjacent in the labyrinth.

We could build a graph based on the description of the labyrinth, but we can actually use the labyrinth itself as a graph. The following code shows how we can explore the labyrinth using depth-first search:

```
def explore(grid, y, x):
    if grid[y][x] != 0:
        return

    print("visit", y, x)
    grid[y][x] = 2

    explore(grid, y-1, x)
```

```
    explore(grid, y+1, x)
    explore(grid, y, x-1)
    explore(grid, y, x+1)

grid = [[1,1,1,1,1,1,1,1],
        [1,0,0,0,0,0,0,1],
        [1,0,1,0,1,1,0,1],
        [1,0,1,1,0,0,0,1],
        [1,0,0,0,0,1,0,1],
        [1,1,1,1,1,1,1,1]]

explore(grid, 1, 1)

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

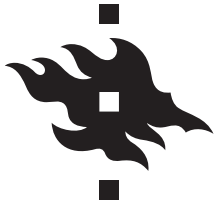
The labyrinth is given as a two-dimensional list, where 0 means a floor square and 1 means a wall square. The function `explore` performs a depth-first search in the labyrinth so that the visited squares are marked with the value 2.

The function `explore` is given the coordinates of a square and it first checks the number in the square. If the number is not 0, the function exits, because then the square is either a wall square or a floor square that has been visited already. If the square is 0, the function marks it visited with the number 2. Then the function continues with recursive calls for the adjacent squares above, below, left and right.

Executing the code produces the following output:

```
visit 1 1
visit 2 1
visit 3 1
visit 4 1
visit 4 2
visit 4 3
visit 4 4
visit 3 4
visit 3 5
visit 3 6
visit 2 6
visit 1 6
visit 1 5
visit 1 4
```

```
visit 1 3
visit 2 3
visit 1 2
visit 4 6
[1, 1, 1, 1, 1, 1, 1, 1]
[1, 2, 2, 2, 2, 2, 2, 1]
[1, 2, 1, 2, 1, 1, 2, 1]
[1, 2, 1, 1, 2, 2, 2, 1]
[1, 2, 2, 2, 2, 1, 2, 1]
[1, 1, 1, 1, 1, 1, 1, 1]
```



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI