Chapitre 7 : graphes et structures de données

I.Data structures

It is a good practice to name your variables, classes, functions using names in English. Your code will be read by people with different backgrounds and also by people who don't speak French, be kind to them.

1.Using lists to represent non-ordered sets

Consider a set of items : $\{a, b, c, d, e, f\}$. This is an *abstract* data structure and can be implemented in a programming language in many different ways. For example :

- A list : `['a', 'b', 'c', 'd', 'e', 'f']`
- A tree : `['a', ['b', 'c'], ['d', ['e', 'f']]]`

The choice of implementation will be different depending on the task. Specifically, the cost, which is expressed as the number of operations, is different depending on the way how data is stored.

We shall define a class implementing a set by using a list.

```
class custom_set:
    def __init__(self, cont=[]):
        self.cont = cont

    def __repr__(self):
        return str(self.cont)
```

Exercice 1a. Implement a function that inserts an element in the given list.

```
def insert(S, elem):
    S.cont.append(.....)
```

Test your function :

```
E = custom_set([])
insert(E, 1)
insert(E, 2)
insert(E, 3)
print(E)
```

What is the cost of insertion of one element in a list of length n ?

Exercice 1b. Implement and test a function that returns the position of the element in the set if it is present, or -1 otherwise.

```
def search(S, elem):
    ....
```

What is the cost of checking whether an element belongs to a list of length n ? (In worst case).

Exercice 1c. Implement your function that deletes an element in a list if it is present in the list (otherwise does nothing).

```
def delete(S, elem):
    ....
```

What is the cost of deletion of one element in a list of length n ?

Now, suppose that we require that a list is always sorted at the end of each operation. This requirement introduces a new data structure : a sorted list. During the insertion, we have to put a new element in the specific position, which is not necessarily at the end of the list. This position can be efficiently found by dividing the list into two parts and comparing the new element with the middle of the array in order to choose the left half or the right half.

Exercice 2. Implement and maintain a class that maintains a sorted list. Let us assume that a list is always initialised as an empty list, and we can only add or delete element one by one. What is the cost of all the operations mentioned in Exercise 1 ?

```
class sorted_set:
    def __init__(self):
        self.cont = []
    def insert(self, elem):
        ......
    def search(self, elem):
        ......
    def delete(self, elem):
        pos = self.search(elem)
        if pos == -1:
            return
        ......
```

2.Binary search tree

It is possible to implement a data structure which makes insertion of one element slower, but accelerates the search and deletion of an element.

Definition. A binary search tree is a data structure that is organised like a rooted binary tree, containing keys in its nodes. If x is the key of the root, then x is greater than all the keys of the left subtree and is smaller than all the keys of the right subtree. Both, the left and the right subtrees, are binary search trees.

Example. Let us describe a binary search tree with 7 elements $\{1, 2, \ldots, 7\}$. If we set its root to 4, then its Python representation will be `[4, [...], [...]]`, where the dots contain the left and right subtrees. Here is a complete example :

```
[4,
    [2, [1], [3]],
    [5, [7, [6]]]
]
```

Verify that this tree is indeed a binary search tree.

Exercice 3. Show that the number of operations required to insert a new element or to search for a new element is $\mathcal{O}(h)$, where h is the height of the tree. Implement and test a class of binary search trees (shortly BST), where each element is a tuple `(key, value)` containing a key and some additional useful information (value) with insertion, deletion, and search of an element.

```
class bst:
    def __init__(self):
        ......
```

The function that searches an element, should return its value if the element is present in a tree, or `None` if not.

Useful to know. In the worst-case scenario, a binary search tree has a form of a line, and the complexity of search and insertion becomes of order n. However, it can be mathematically proven that if such tree is formed uniformly at random, then its height is $O(\log n)$. Furthermore, there exist other, more sophisticated structures that guarantee $O(\log n)$ complexity in worst cases, but it takes more time to explain how they function.

3.Python's set and dictionary.

A dictionary is a database, or a set of elements, where every element can be searched by a key, and contains additional value. Binary search trees are one of the numerous possible implementations of dictionary. The Python language has its own implementation of dictionary, and for the purpose of this exercise, we are not very interested in how exactly it works. The only thing we know is that its implementation is efficient.

Exercice 4a. Create your first set and dictionary and print them :

```
instruments = {'piano', 'drum', 'violin', 'flute'}
prime_factorisations = {
    6: (2, 3),
    2: (2,),
    15: (3, 5),
    2022: (2, 3, 337)
}
```

Exercice 4b. Familiarise yourself with how sets and dictionaries work by accessing their elements. By using autocomplete, observe other available methods.

```
print('drum' in instruments)
print('guitar' in instruments)
print(7 in prime_factorisations)
print(prime_factorisations[6])
print(prime_factorisations.pop(6))
print(prime_factorisations)
prime_factorisation[14] = (2, 7)
print(prime_factorisations)
instruments.add('guitar')
instruments.remove('piano')
print(instruments)
```

Note that sets and dictionaries support generators too :

```
print({i: i**2 for i in range(5)})
```

Useful to know. If you implement your own class, you can overload the operator "`in`" according to your needs :

```
MyClass.__contains__(self, item)
```

II.Excursions on graphs

1.A notion of graph

A graph is an abstract data structure, just like a set, and again, it can be implemented in many different ways, depending on the specific problem that you need to solve. Depending on the implementation, some of the operations will be faster, and some will be slower.

We will work with two types of graphs : simple and directed. Variations also include multigraphs and directed multigraphs.

Informal definition. A *graph* is a set of points connected with lines that are called *edges*. A graph is *directed* if its edges have directions. A *multigraph* is a graph where loops are allowed and multiple edges between two vertices are allowed.

Formal definition. A *graph* is a pair (V, E), where $E \subset \{\{x, y\} \mid x, y \in V,\, x \neq y\}$. A *directed graph* is a pair (V, E), where $E \subset \{(x, y) \mid x, y \in V,\, x \neq y\}$. A *multigraph* is a pair (V, E), where $E = (e_1, e_2, \ldots, e_m)$ and $e_i \in \{\{x, y\} \mid x, y \in V\}$.

Exercice 5. How many different graphs with n vertices are there ? How many different graphs with n vertices and m edges ? How many digraphs ?

2.Representations of graphs

There are three main approaches to storing graphs in computer memory.

- **Adjacency matrix.** If a graph has n vertices, then it is stored as a matrix A of size $n \times n$ with zeros and ones. If $A_{ij} = 0$ then there is no edges between vertices i and j, otherwise $A_{ij} = 1$.
- **Incidence matrix.** If a graph has n vertices and m edges, then it is stored as a matrix A of size $n \times m$, and we have $A_{ij} = 1$ if and only if jth edge is incident to ith vertex.
- **Adjacency list.** For each vertex v in graph, there is a set of vertices that are connected to v. A graph is stored as a collection of these sets.

These approaches can be adapted to directed graphs and multigraphs. The most preferred approach is the adjacency list because it reduces the cost of keeping additional zeroes if the graph does not have too many edges.

Exercice 6. For a graph with a given adjacency matrix,

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

write its incidence matrix and adjacency list. Implement a function that converts an incidence matrix into an adjacency list and check your computations :

```
def incidence_mat_to_adjacency_list(mat):
    """This function converts an incidence matrix
    to adjacency list
    """
    n = len(mat)
    m = len(mat[0])
    # Create a list of empty sets for each vertex
    result = [set([]) for _ in range(n)]
    ......          # fill in the elements of ‘result‘
    return result
```

Useful to know. Similarly to sets and dictionaries, you can imagine that the vertices and edges of graphs are the keys and they can carry some additional information, for example, edge length. A *path* in a graph is a sequence of vertices connected by edges. If the edges have lenghts, then the length of a path is a sum of the lengths of edges. One can search, for example, for a path with minimal length between two given vertices, just like you are searching for the fastest route in Google Maps.

3.Depth-first search (dfs)

Let us solve the following problem. Given a graph as an input, check if there is a path between two vertices of a graph. This is equivalent to checking whether two nodes x and y are in the same connected component.

This problem can be solved by using *depth-first search* algorithm. This algorithm keeps track of whether a node is already visited or not. Initially, all the nodes are unvisited. Then, we visit the vertex x and recursively launch the search to all of its neighbors. Each time we check the list of all the neighbors and visit all the unvisited nodes, recursively, one by one. When the algorithm terminates, we check whether y is visited or not.

Exercice 7. For the purpose of this exercise let us assume that the labels of the nodes of a graph are distinct numbers from 1 to n and it is given as an adjacency list. Implement a function that returns `True` is there is a path between two vertices x and y in a graph, or `False` otherwise.

```
def is_path(graph, x, y):
    n_nodes = len(graph)
    visited = [False for _ in range(n_nodes)]
    def dfs(v):
        visited[v] = True
        for u in graph[v]:
            ......
    dfs(x)
    return visited[y]
```

Exercice 8. Now, let us assume that the labels of the nodes are not necessarily numbers, but can be arbitrary objects, for example, strings. Modify the above code to make it work for this more general class of graphs. Hint : instead of introducing an array of `True-False` for visited vertices, start with an empty set and add vertices that are visited.

4.Breadth-first search (bfs)

Now we are going to solve a more complicated problem : determine a shortest path between two given vertices of a graph. In this exercise we are asssuming that each edge has length 1. This problem cannot be solved with the dfs algorithm, and we need to go more slowly :

- First, write down the list of all the neighbors of a vertex ;
- Second, add all the neighbors to a queue and process every element in the queue by recursively using a bfs ;
- For every element in the queue, when exploring its neighbors, don't jump into the search immediately, but add the neighbours to the end of the queue.

Before you start implementing the bfs, you need to familiarise yourself with a queue datatype. Python has a double-ended queue (deque) in a dedicated module, and supports adding and removing an element at both sides in $O(1)$ operations. Using a queue instead of a list is important, because removing the first element of the list takes n operations, where n is the length of the list.

Exercice 9. Familiarise yourself with the deque. By using autocomplete, observe other available methods.

```
from collections import deque
Q = deque([1,2,3])
print(Q)
print(Q.popleft())
print(Q.popleft())
print(Q)
Q.append(4)
print(Q)
if Q: print("Q is not empty")
print(Q.popleft())
print(Q.popleft())
if not Q: print("Q is empty")
```

Exercice 10a. Observe that if the algorithm starts with a node x, then first there will be processed nodes that are at distance 1 from x, then all the nodes that are at distance 2, etc. If we keep the current distance during the search, then the first time we visit the node y we will know the distance between x and y.

Exercice 10b. Implement a Python function that visits all the nodes by using a bfs procedure and returns the list of visited nodes.

```
from queue import deque
def connected_component(graph, x):
    visited = set([])
    Q = deque([x])
    while Q:
        v = Q.popleft()
        visited[v] = True
        ......          # visit all unvisited neighbors
    return visited
```

Exercice 10c. Modify the above function : add a variable that tracks the current distance to the initial vertex, and compute the shortest distance between x and y.

Hint : keep the "current" queue for the current distance and the "next" queue for vertices with distance increased by 1. Once you reach the end of the current queue, increase the distance by 1 and swap the queues.

```
def distance(graph, x, y):
    visited = set([])
    Q_current = deque([x])
    Q_next = deque([])
    result = 0
    while Q_current:
        v = Q_current.popleft()
        visited[v] = True
        ......          # visit all unvisited neighbors
        if not Q_current:
            ....        # increase the distance by 1
            ....        # swap the queues
    return result
```

Exercice 10d. Modify the code of exercise 10b : for each of the visited vertices keep track of its "parent" in the order of visit. When the vertex y is reached, compose the path by recursively following the parents, and return the resulting path.

```
def shortest_path(graph, x, y):
    visited = set([])
    Q = deque([y])
    parents = {}   # empty dictionary
    parents[y] = y

    while Q:
        v = Q.popleft()
        visited[v] = True
        ......     # visit all unvisited neighbors
        ......     # v becomes a parent
    if x not in visited:
        return None

    result = [x]
    v = x
    while parent[v] != v:
        ......
    return visited
```

5.Dijkstra's algorithm

We can apply a very similar bfs-based approach when the edges have a length to solve the problem of finding the shortest path between the two nodes. Now we store a graph (again) as a dictionary of nodes, where each node has a dictionary of neighbors. Each neighbor is represented as a pair, containing the key (the name of the neighbor) and the value (the length of the edge). Here is an example of graph with distances between some of the cities of France.

```
France = {
    'Paris': {'Dijon': 314, 'Tours': 239, 'Angoulême': 452},
    'Dijon': {'Lyon': 196, 'Paris': 314},
    'Tours': {'Paris': 239},
    'Angoulême': {'Paris': 452},
    'Lyon': {'Dijon': 196}
}
```

The new algorithm lauches a bfs on the vertices and keeps the list of visited vertices. Initially, all the vertices are unvisited. We are solving a more general problem : for a given vertex x, find the distances towards all the vertices. Initially, all the distances are set to infinity (10^10) and will be updated if a shorter path is found.

Exercice 11. Complete the code.

```
def distance(graph, x, y):
    visited = set([])
    dist = {v: 10**10 for v in graph}
    Q = deque([x])
    while Q:
        v = Q.popleft()
        for u in graph[v]:
            if visited[u]:
                continue
            dist_to_u = dist[v] + graph[v][u]
            if ......
        visited[v] = True
    if not visited[y]:
        return None
    return dist[y]
```