Introduction to algorithms Fall 2017

IFT2125-6001 TA: Maëlle Zimmermann

Demonstration 5

1

Question: In graph theory, an isomorphism of graphs G and H is a bijection
f between the vertices of G and H

$$f: V(G) \rightarrow V(H)$$

such that if two u and v are adjacent in G, then f (u) and f (v) are adjacent
in H. Give an algorithm that randomly generates two graphs and compares them
by number of edges, number of vertices, and sequence of degrees. Is this enough to verify
if the two graphs are isomorphic?

Solution: Here are the algorithms:

```
import random

# generates a random graph impingement matrix
random_graph def (n):
    g = [[random.randint (0, 1) for i in range (n)] for j in range (n)]
    for i in range (n):
        g [i] [i] = 0
    for i in range (n):
        for j in range (i, n):
            g [j] [i] = g [i] [j]
    return g

# compare two graphs
def graph_iso (g1, g2):
    if len (g1) = len (g2)!:
        return False
    if sum (sum (x) for x in g1) = sum (sum (x) for x in g2!):
        return False
    seq_deg_g1 = [sum (x) for x in g1]
```

```
seq_deg_g2 = [sum (x) for x in g2]
if sorted (seq_deg_g1) = sorted (seq_deg_g2!):
    return False
return True
```

This algorithm is not enough to check if two graphs are isomorphic, even if it allows to detect certain cases where they are not. Here is a counterexample:

These two graphs have the same number of edges, vertices and the same sequence of degrees [3,2,2,2,2,1,1,1] but are not isomorphic.

On the other hand, if one counts for each vertex the degrees of its neighbors, one can see that the graphs are not identical. Indeed, two of the vertices to degree 1 of the first graph have a neighbor of degree 2, while in the second graph only a vertex to degree 1 has a neighbor of degree 2. This is the idea behind the Weisfeiler-Lehman algorithm which calculates for each vertex the degrees of vertices at distance 1, then at distance 2, etc. Although this algorithm works better, it still fails on certain types of graphs. In fact, there is no known polynomial time algorithm for this probl`eme.

2

Question: Give an efficient implementation of disjoint sets and analyze the complexity in time.

Solution: Suppose we have k objects numbered 1 to ak and we want group them in disjoint sets, that is to say that at any moment, each object is in exactly one set. For example, if k = 10, the objects can be partitioned as well.

$$\{1,3,7\}, \{2,5,6,10\}, \{4,9\}, \{8\}.$$

Each set is associated with a label. For example, one can choose by conven- to denote each set by its smallest element, in which case {2,5,6,10} is labeled with 2.

We are interested in analyzing the following operations:

- find (x): returns the label of the set containing x,
- merge (a, b): merges the sets labeled by a and b.

For example, above find (6) returns 2, and after merge (1,4) we get the partition

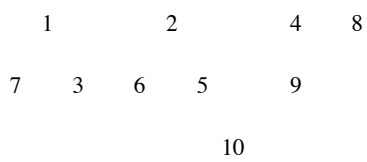
{1,3,4,7,9}, {2,5,6,10}, {8}.

We want to represent this problem efficiently on a computer.

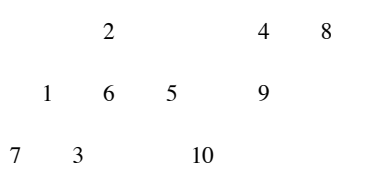
We could consider an implementation where each element is associated with the label of its set, for example here the initial partition would be represented by the list S = [1,2,1,4,2,2,1,8,4,2]. In this case the operation find (x) would take a constant time, but the merge operation (a, b) would not be efficient and take a time in Θ (k) in worst case scenario. For example, by merging two sets of size k / 2, change k / 2 tags.

Consider an implementation where the sets are represented by arborescences and define that the label of a set is the number at its root. In Each element is associated with its parent except roots that are themselves. In addition, we will also store, for each x, the height of the root tree of x. The merge of two trees is done by merging the smallest to the largest.

For example, the above partition could be represented by the following tree which would give the list S = [1,2,1,4,2,2,1,8,4,5]:



After the merge operation (1,2), we obtain S = [2,2,1,4,2,2,1,8,4,5]:



Thus the disjoint sets would be represented by a tuple D = (S, H) where H is the list that contains the heights of the trees. Here is a python implementation of operations find (x) and merge (a, b) according to this idea:


```
def find (D, x):
    S, D = D
    r = x
    while S [r] = r!:
        r = S [r]
    return r

def merge (D, a, b):
    S, H = D
```

```
if H [a] == H [b]:
    H [a] = H [a] + 1
    S [b] = a
elif H [a]> H [b]:
    S [b] = a
else:
    S [a] = b
```

We can show that in this case the complexity of find is in Θ (1) in the worst case and in Θ (log k) in the worst case, because the height of the trees never exceeds log k, taking care to merge the smallest `to the highest` at each merge. The complexity of merge is in Θ (1) in the best and worst case. For more details, read Brassard and Bratley's section 5.9 (pp. 175-180).

3

Question: Apply Kruskal's algorithm to find a tree underlying minimum weight on the next graph, where the weight is shown next to each edge.

4

Solution: The Kruskal algorithm allows to find a tree underlying weight of a connected graph. The algorithm initializes an empty set T and runs the edges (u, v) in ascending order of weight. If the u and v points connected by the edge are in two different connected components, the edge is added to a T. By executing the algorithm we obtain:

Iteration	(u, v)	Set of edges T	Related Components D
0	-	-	{1} {2} {3} {4} {5} {6} {7}
1	(1,2)	{(1,2)}	{1,2} {3} {4} {5} {6} {7}
2	(2,3)	{(1,2), (2,3)}	{1,2,3} {4} {5} {6} {7}
3	(4,5)	{(1,2), (2,3), (4,5)}	{1,2,3} {4.5} {6} {7}
4	(7.6)	{(1,2), (2,3), (4,5), (7,6)}	{1,2,3} {4,5} {6,7}
5	(1,4)	{(1,2), (2,3), (4,5), (7,6), (1,4)}	{1,2,3,4,5} {6,7}
6	(2.5)	{(1,2), (2,3), (4,5), (7,6), (1,4)}	{1,2,3,4,5} {6,7}
7	(4.7)	{(1,2), (2,3), (4,5), (7,6), (1,4)	{1,2,3,4,5,6,7}
...			
12	(5,6), (1,2), (2,3), (4,5), (7,6), (1,4), (4,7)		{1,2,3,4,5,6,7}

Thus the minimum weight subtender tree contains the edges in T and is of weight 17.

4

Question: Implement the Kruskal algorithm and analyze its execution time in the worst case.

Solution: Here is a python implementation based on the implementation of sets disjoined from the previous financial year:

```
def Kruskal (V, E)
T = []
def init (k):
    return (range (k) [0] * k)
D = init (len (V)) # initiates all disjoint (connected components)
def weight ((u, v, c)):
    return c

# loop on edges sorted in ascending order
for (u, v, c) in sorted (E, key = weight):
    i = find (D, u) # is connected component of u
    j = find (D, v) # is connected component of v
    if i = j #! if u and v are in the same connected component
        merge (D, i, j) # merge sets
        T.append ((u, v)) # add the ridge
return T
```

Let n = | V | and a = | E |, then:

- The initialization of disjoint sets of vertices is done in time Θ (n),
- The sorting of edges is done in time Θ (alog a),
- There are loop towers, so 2a calls to a find,
- After n - 1 mergers there is only one set left, so there are - 1 calls to merge.

Since each call to find takes a time in Θ (log n), and every call to merge takes a time in Θ (1), the total execution time of the Kruskal algorithm is in Θ (n + alog a + 2alog n + n - 1) = Θ (max {n, alog a, 2alog n, n - 1}). However, graph is related, we have

- n - 1 ≤ a because it takes at least n - 1 edges to connect all the vertices,
- a ≤ n (n-1) / 2 because it is the maximum number of edges that can be included in a graph to angels.

Thus, the execution time of the algorithm is in Θ (max {alog a, alog n}) = Θ (alog n) because log a ≤ 2 log n log n = 2.

5

Question: A server has clients to serve and can only serve one at a time. The weather of service required by each customer is known in advance: customer i takes a long time t_i. We try to minimize the average waiting time of clients in the system. Give an algorithm for this problem and show that it works.

Solution: This is equivalent to minimizing

$$T = \sum_{i=1}^{\text{not}} \text{time in the client i system}$$

Idea: Serve customers in increasing order of time, ie first which require the least amount of time.

If we formulate this idea in the form of an algorithm, this corresponds to formulating an algo- voracious ritual that agendates customers one after the other without ever returning on its choices precedents. At each step, the algorithm simply chooses the following client as having the smallest service time required among the remaining customers. The algorithm when all customers are on call, that is, tation. For example, the algorithm can be formulated as follows:

```
def ord (t):
    def time (i): return t [i]
    Customers = range (len (t))
    return sorted (customers, key = time)
```

Let us show that this algorithm gives the optimal solution. Let $P = p_1 p_2 ... p_n$ a ordon- n optimization of n clients.

Let $s_{i\,i}\,t_p$ = processing time of the i_{th} customer `served in the order defined by P. If customers are served in the order defined by P, then the total time required is

$$T(P) = s_1 + (s_1 + s_{2)} + (s_1 + s_2 + s_{3)} + \ldots$$
$$Ns = {}_1 + (n - 1)\, s_2 + (n - 2) + s_3 \ldots$$
$$= \sum_{i=1}^{not} (n - i + 1)\, s_i$$

Suppose by the absurd that P does not order customers in increasing order of time treatment. Then there exists a <b such that $s_{a>}\, s_{b.}$ Let P be the scheduling where these clients are interchanged in P. The total time required in the order defined by P is:

$$T(P) = (k - 1 + a)\, s + {}_b\, (k - b + 1)\, s_a + \sum_{i=1,\, i=a,\, b}^{not} (k - i + 1)\, s_{i.}$$

So,

$$T(P) - T(P) = (k - a + 1)\,(s_a - s_{b)} + (k - 1 + b)\,(s_b - s_{a)}$$
$$= (K - a + 1)\,(s_a - s_{b)} - (k - 1 + b)\,(s_a - s_{b)}$$
$$= (B - a)\,(s_a - s_{b)} > 0.$$

Therefore the total time using the scheduling P is smaller than that obtained in using scheduling P. This contradicts the fact that P is the optimal scheduling. Thus, assuming that P does not order customers in ascending order, contradiction. Thus, the optimal ordering P order the clients in order growing time.