HW1 - Algorithm Design, Sapienza

Michael Capponi 1711759

2018-12-1

1 Ex 1

The goal of this exercise is to maximize the distance $d(x, c)$ between any point x of our metric space (X, d) and its closest center c, where c belongs to a k-large set of centers and k $\in \{1, ..., |X|\}$.
So, to solve this exercise I will exploit a k-center clustering approach and the property of triangle inequality. This approach results to be an approximation of the real problem that makes it possible to solve the k-clustering problem in a polynomial time by using a greedy algorithm. K-clustering problem is indeed a NP-hard problem and it is not possible to find an exact poly-algorithm to solve it. But in our case, the exercise decreases the number of possible cases that we can have when choosing a new center because there are only three possible distances, so I will prove that the approximation algorithm will result in the exactly real case.
Since we have to maximize our objective function for any such k, let's repeat for each k these steps:

- Choose an arbitrary point into X, this will be the first center, let me call c_1 and add to the set of centers C.
- For every point x $\in X$ let's compute the distance $d(x, c_1)$.
- Pick the point with maximum distance from c_1. This will be the second center c_2 and add it to C.
- The next step is to pick another point that satisfies the following constraint: it has to be the farthest point beetwen c_1 and c_2. This will be the third center, c_3.
- Continue in this way until the k centers are found and every time a center is chosen, obviously we assign every point of X to its closest center in C.

Let's analyze the running time of this algorithm and why it maximizes the objective function:
in the i-th iteration, we choose a new center c_i such that it satisfies
$r_{i-1} = max_{x \in X}[d(x, C_{i-1})]$

where C_{i-1} is the set of centers containing all the centers between c_1 and c_{i-1} and r_{i-1} is the radius of clustering and is computed in every iteration.
This step is repeated k times. So, in each iteration, the distance from all point in X to the set of current centers is calculated.
Now, we know that for a point x $\in X$, we have:

$$d_i[x] = d(x, C_i) = min(d(x, C_{i-1}), d(x, c_i)) = min(d_{i-1}[x], d(x, c_i))$$

If we maintain a single variable d[x] that store the current distance between a point x $\in X$ and its closest center in the current center set C, in each iteration we only need to compute $d(x, c_i)$. Hence, the single i-th iteration takes $O(n)$ time to choose the center c_i and since we have k iterations, the total running time is $O(nk)$.
To prove the correctness of this approximation I exploit a theorem that says:

- Given a set of n points P $\subseteq X$, where X is a metric space (X, d), the greedy K-center algorithm computes a set K of k centers, such that K is 2-approximation to the optimal k-center clustering of P. That is:
  - $r^K(P) \leq 2r^{opt}(P, k)$

Now, we have 4 possible cases according to problem constraint which says that the distance d(x,y) between 2 points x,y $\in X$ must be only: 0 if x = y, 1 or 3. So we have:

1. Let k_i be the next center found by the approximation algorithm and let's suppose its distance from the current set center is equal to 1 while the real distance found from the optimal execution is 3. This can't be possible because the greedy algorithm takes as next center the one which maximizes the distance to the current set center.

2. Let k_i be the next center found by the approximation algorithm and let's suppose its distance from the current set center is equal to 3 while the real distance found from the optimal execution is 1. This means that, according with inequality of the theorem I'd written above:

   - $r^K(P) \leq 2r^{opt}(P, k) \rightarrow 3 \leq 2 * (1) \rightarrow 3 \leq 2.$

   This is clearly a contradiction of the hypothesis.

3. The other two cases are those where approximation center matches with the optimal center. This is for definition the optimal but also the only solutions, so I prove that the algorithm provides only the optimal solution according to the problem's constraint.

At last, considering we have to analyze all possible k between 1 and n to be sure about how to cluster the metric space, so we try for n possible k values where n is X's cardinality, the cost of the greedy algorithm inwill be:

$$\mathbf{O(kn) = O(n^2) = O(|X|^2)}$$

2 Ex 2

The exercise requires an algorithm to find the minimum number of cameras to cover all checkpoint. Of course, I can find an algorithm running in polynomial time which takes as input parameters a set of streets S, a set of avenues A and a set of checkpoints C whose instances are tuples c = (s, a) where a is an instance of A and s an instance of S.
Let's first explain how it is possible deal with this type of problem: consider the intersection of among streets and avenues as a grid. The goal is to find a subset of the union between streets and avenues such that if I color its lines I will cover all the checkpoints. What I can exploit of this configuration is to consider this grid as the adjacency matrix of a bipartite graph. A bipartite graph is a particulare graph in which one can consider two different types of vertices and where the edges link only two vertices belonging to different types.
So I can consider a vertex for each street and define a first type of vertices, S, and then consider another vertex for every avenue and define the second type of vertices, A. At last, I will link, through a directed edge from S vertex to A vertex, every pair of vertices corresponding to a pair of streets and avenues in which intersection there is a checkpoint.
Then I transform this bipartite graph to apply the "Max flow formulation": given my bipartite graph G, I add two nodes s and t such that:

- link all vertices belonging to type S with a directed edge starting from s with unit capacity.
- link all vertices belonging to type A with a directed edge starting from each vertex of A and ending in t with unit capacity.

And what I obtain is a new graph, let's call it G'.
Now, by exploiting some well-known theorem I can prove that the algorithm I'm going to use works correctly and in a poly-time. First consider the "Konig's theorem":

Given a bipartite graph G(U,V,E) find a vertex set S $\subseteq U \cup V$ of minimum size that covers all edges, i.e. every edge (u,v) has at least one endpoint in S, that is u $\in S$ or v $\in S$ or both.

Let now M be a maximum cardinality matching in G, that is an edge set $M \subseteq E$ such that no two edges of M have the same endpoint. Since every edge of M must have at least an endpoint of S, the size of maximum matching M is a lower bound on the size of the minimum number of cameras to use to cover all checkpoints. So, Konig's theorem states that the maximum matching cardinality is equal to the minimum cameras cardinality. I will exploit this. Let's see how.
I can find the maximum cardinality of a matching by using the already mentioned "Max flow formulation". This states that the max cardinality of a matching in G is equal to the value of max flow in G'.
Hence, using Konig's theorem and the Max flow formulation I stated that the number of minimum cameras I have to use to cover all checkpoints corresponds

to the value of max flow in G' which is built starting from G nodes and edges. Let me use the Ford-Fulkerson algorithm to find the max flow value of G'.
The algorithm is quite finished: indeed, Ford-Fulkerson algorithm designs some augmenting path to find the max flow. So, Ford-Fulkerson will output the maximum matching subgraph, that is a subset (let me call M) of $U \cup V$.
Let's consider now the subset Z defined as: $Z = S \setminus M(S)$, that is the vertices of S which are not matched in M. Now, I will find the set CAM of streets or avenues where to place the cameras and at the same time prove the correctness of the algorithm.
Let's execute these final steps:

- Add to Z all vertices connected to some vertex of Z through an "alternating path" that is a path which link two vertices with alternating edges from the matching set M and edges not from M.
- Now, find the set S defines as: $CAM = (S \setminus Z) \cup (A \cap Z)$

The construction of Z implies that for every edge $(u, v) \in M$, if $v \in Z$ then $u \in Z$ as well. And if $u \in Z$, since u was not initially in Z among the non matched vertices in S, it must be that u was added to Z by following some matched edge, which means that $v \in Z$ as well.
This implies that for every edge $(u, v) \in M$ of the matching either both endpoints are in Z or none. Hence every edge from the matching has exactly one endpoint in CAM, and $|CAM| \geq |M|$.
Now I will show that CAM covers all edges of the graph. Let (u,v) an arbitrary edge. If $u \notin Z$, then the edge is covered. From now on assume that $u \in Z$. If (u,v) is not an edge from the matching then by maximality of Z the vertex v has to be in Z, and hence $v \in CAM$. But if (u,v) is an edge from the matching, then by the previous observation v belongs to Z and hence to CAM as well. This shows that CAM is a vertex cover, which implies by the first observation $|CAM| \leq |M|$, from which we can conclude $|CAM| = |M|$.

Algorithm 1 Find the subset with minimum cardinality of cameras to cover the checkpoint

```
procedure PLACE_CAMERAS($S, A, C$)
    $s \leftarrow$ newNode()
    $t \leftarrow$ newNode()
    $G \leftarrow$ newEmptyNodesSet()
    $E \leftarrow$ newEmptyEdgesSet()
    $G \leftarrow$ AddNode(s)
    $G \leftarrow$ AddNode(t)
    for all $v \in S$ do
        $E \leftarrow$ AddEdge(s, v, 1)
        $G \leftarrow$ AddNode(v)
    end for
    for all $v \in A$ do
        $E \leftarrow$ AddEdge(v, t, 1)
        $G \leftarrow$ AddNode(v)
    end for
    for all $c \in C$ do
        $u \leftarrow$ getStreet(c)
        $v \leftarrow$ getAvenue(c)
        $E \leftarrow$ AddEdge(u, v, 1)
    end for
    $S\_S, S\_A, M\_E, n \leftarrow$ FordFulkerson(G, E, s, t)
    # S_S, S_A: subset of matching streets and avenues; M_E: subset of matching edges;
    # n: minimum number of cameras, equal to $|M\_E|$.
    $Z \leftarrow S \setminus S\_S$
    $Z \leftarrow$ Find_Vertices_Through_AltPath(Z, G, M_E)
    $CAM \leftarrow (S \setminus Z) \cup (A \cap Z)$
    return CAM
end procedure
```

The running tume of this algorithm is given by its more expensive operation. This is the max-flow computed by the Ford-Fulkerson algorithm. This runs in $O(|E|f)$ time, where f is the value of max-flow it has found. In the worst case of our problem, where a checkpoint is placed on each possible intersection, f will be equal to $min\{|U|, |V|\}$ and $|E|$ will be the number of checkpoint, so $|U|*|V|$, the running time will be $O(|U| * |V| * min\{|U|, |V|\})$ So the running time founded is polynomial in the number of streets and avenues.

3 Ex 3

We are given a problem, let's call it problem "A" and we have to show that it is a NP-Complete problem. A problem X is NP-Complete if for every problem Y in NP, $Y \leq_p X$. So, in order to prove that A is NP-Complete there are two steps to fullfil:

1. Show that A is an NP problem.
2. Reduce an already known NP-Complete problem to A.

About the first point I have to find a certifier which, given a possible solution (s) of A verify in poly-time that it is really a solution.
In order to be a valid solution for problem A, s must verify the following requirements:

- Number of male invited friends of s is the same of female invited friends' number.
- Given an already known solution t of A called "certificate", must be that $obj\_func(s) \geq obj\_func(t)$

So a good certifier will be:

Algorithm 2 Find a certifier for problem A

```
procedure CERTIFIER($s, t$)
    if $|s \cap M| \neq |s \cap F|$ then
        return False
    end if

    if $\frac{\sum_{(x,y)\in s} w(x,y)}{|s|} >= \frac{\sum_{(x,y)\in t} w(x,y)}{|t|}$ then
        return True
    else
        return False
    end if
end procedure
```

Now, the reduction.
Before introducing an already known NP-Complete problem, let me just transform our optimization problem in a decision problem. This lets to easily handle the reduction. To do this I have to show that if this problem, let's call it C, is NP-Complete, then also A will be NP-Complete.
The C problem will have the same constraint of problem A about the same number of male and female invited friends but now we are given a value x and we have to decide wheter the objective function for any solution s satisfies obj_func(s) = x. We can transform this problem C to A by simply iterate a call to C for every possible $x \in (1, ..., 2min\{|M|, |F|\})$ or by using binary search. So

if C is NP-Complete, then also A must be NP-Complete. Note that for C exists a simple certifier that just verify that a given solution s respects the constraint: $|s \cap M| = |s \cap F|$.
I would like to use the NP-Hard problem of decision k-clique, that is, given a graph and any value of $k \in 1, ..., n$, decides wheter exists a k size clique.
Let's call this problem B. If I show that $B \leq_p C$ then I have shown that C in NP-Complete and so also A it is.
So, given any instance of B , I(B) and so a graph with n nodes and a number k, the first step is to transform it in an instance for C: I(C).

1. Let's take the input graph G and transform it in a second graph G' which contains G and also other k nodes connected in a k-clique by $\frac{k(k-1)}{2}$ further edges.

2. Let the set made up from G nodes represents males friends.

3. Let the k-clique represents the females friends.

4. Let's choose as the given number x, $x = \frac{k-1}{2}$. Why this number? Well, this is the maximum density the set of females friend can have. According to the problem C constraint, it will be also the maximum density accepted for males friends' set.

This procedure will take $O(|G'.V| + |G'.E|)$ time. So it is a poly-time and satisfies the contraint of NP-Complete proof.
Now what we have to do is to run our ALG_C which solves the A problem (of course in a not poly-time) and take the output.
Let's analyze it.
In order to obtain the value x, the ALG_C is forced to take all female's friends. Indeed if it take less of k female's friends it will not be able anymore to reach the value x in order to respect also the second constraint: $|I \cap M| = |I \cap F|$.
So the algorithm will take exactly k female's friends with a related score of $k(k-1)/2$. Now, again for the problem C constraint, the algorithm will also take exactly k male's friend. This means that, in order to stafisy obj_func = x, the male's score w must satisfy: $\frac{w+k(k-1)/2}{2k} = \frac{k-1}{2}$ Solving this equality we obtain $w = \frac{k(k-1)}{2}$. But this means that also male's friend will be a k-clique node.
Hence, the value x could be reached if and only if also the male's friends graph contains a k-clique. In this case the decision problem C will return a True value with the friend's subset found, otherwise it will return a False value.
Transforming back this solution, let's call it I, to B problem's instance, so after deleting from G' nodes and edges related to female friends (this will take poly-time), we are able to check that if a k-clique exists for the instance graph G. In this way the proof of C's NP-Completeness if finished and consequently, also A is a NP-Complete problem.

4 Ex 4

4.1 Ex 4.1

We are given a set of tasks which are distributed on a subset of time instants $1, ..., T$. We also can have more than one task for each time instant or even no one task.
My solution exploits dynamic programming in the fact that it stores the results of minimum cost in a time t1 = i and re-use it in next instants of time t2 = i + j , $t1, t2 \leq T$. This approach can be considered a memorization approach. My optimal strategy consists of cutting all the possible choices which never could be chosen because they are more expensive than others. For example, if at time t1 = i - 1 I've hired a worker, at time t2 = i I never choose to outsource a freelance and maintaining that worker, because in that case I'm paying more than simply maintaining a worker: $s < s + f(t2)$.
At the first instant in which I have a task, let's call t0, I have not any hired worker or outsourced freelance, so I have only two choices:

- Hire a new worker and pay $A + s$.
- Outsource a freelance and pay $f(t0)$.

Now, in the instants following t0, the choices I can do are only 4:

- At time t1 = i - 1 I have a worker and now I choose to continue to pay him, so at time t2 = i, what I'm paying considering all previous instants is $tot\_cost(t1) + s$.
- At time t1 = i - 1 I have a freelance and now I choose to hire a new worker, so at time t2 = i, what I'm paying considering all previous instants is $tot\_cost(t1) + A + s$.
- At time t1 = i - 1 I have a worker and now I choose to outsource a freelance, so at time t2 = i, what I'm paying considering all previous instants is $tot\_cost(t1) + F(t2) + L$.
- At time t1 = i - 1 I have a freelance and now I choose to outsource again freelance, so at time t2 = i, what I'm paying considering all previous instants is $tot\_cost(t1) + F(t2)$.

In each time instant I will have a single worker or a single freelance: so I compute the minimum cost of hiring a worker and the minimum cost of outsourcing a freelance, so that in the next time instant I reduce the problem simply to have a worker or a freelance and this will lead me again to 4 possible choices to consider and so on until to the time instant T.
So, I can consider two possible paths through the time: one that consider at each time to outsource a freelance and one that consider at each time to hire a worker. The recursive formula will be like this:

- $$Cost(w,i) = \begin{cases} A + s & \text{if i = 1} \\ min\{Cost(w,i-1)+s, Cost(f,i-1)+A+s\} & \text{if i > 1} \end{cases}$$

- $$Cost(f,i) = \begin{cases} f(1) & \text{if i = 1} \\ min\{Cost(w,i-1)+L+f(i), Cost(f,i-1)+f(i)\} & \text{if i > 1} \end{cases}$$

I will call both this functions with a given instant T and the take the minimum:
$min\_cost = min\{Cost(w,T), Cost(f,T)\}$

The correctness can be proved for induction:

- At the base case we have only to choices, hire a worker ot ousource a freelance. The minimum is trivially given by the less expensive cost.
- At the general time t, assuming that the algorithm is correct at time $t-1$, we have the tot_cost(t - 1) plus the cost of our next choice. It works by just taking the one with the minimum cost.

In this case the running time will be proportional to T because at each instant t I have only 4 possibilities to consider and for each of them I'm not computing all previous steps until to the first time instant each time but simply store the previous one and re-use it according to the dynamic programming approach. This states that running time is $O(T)$.
At the end, I will simply choose the minimum between what I've computed following the "worker path" and what I've computed following the freelance path".

```python
import random

T = int(input("Give a number of instants in which are distributed
    the tasks: "))
N_task = int(input("Give a number of tasks: "))

print("\nCreating random outsourcing costs between 1 and 10...")
F_t = []
for i in range(0,N_task):
    F_t.append(random.randint(1,10))

Task = []
print("Creating the pair instances of Tasks...")
for i in range(1,N_task):
    Task.append((random.randint(1,T),F_t[i]))
print()
A = int(input("Give a cost tor hiring a worker: "))
s = int(input("Give a cost of worker's salary: "))
L = int(input("Give a cost to firing a worker: "))
print()
F_C = [0]*T
for (t,cost) in Task:
    F_C[t-1] += cost

Workers_Execution = {}
Freelancers_Execution = {}

def frl_cost(t_i):
    if t_i in Freelancers_Execution:
        return Freelancers_Execution[t_i]
    elif t_i == 0:
        return 0
    else:
        wrk = wkr_cost(t_i - 1) + F_C[t_i - 1] + L
        frl = frl_cost(t_i - 1) + F_C[t_i - 1]
        Freelancers_Execution[t_i] = min(frl, wrk)
        return Freelancers_Execution[t_i]

def wkr_cost(t_i):
    if t_i in Workers_Execution:
        return Workers_Execution[t_i]
    elif t_i == 0:
        return A
    else:
        wrk = wkr_cost(t_i - 1) + s
        frl = frl_cost(t_i - 1) + A + s
        Workers_Execution[t_i] = min(wrk, frl)
        return Workers_Execution[t_i]



def tot_cost(T):
        min_cost = min(wkr_cost(T), frl_cost(T))
        return min_cost

print("The mininmum cost computed is: ", str(tot_cost(T)))
```

4.2 Ex 4.2

Now, we are given a set $W_t \subseteq W, |W| = k$ for every instant, that is a set of skills required from any task. We have to decide for each skill in W_t if to hire a worker or to outsource a freelance.
This means that this problem could be solved in the same way of the previous one:

- At time t1 = 1, we have not freelances or workers, so we just have to decide for each skill what to do between these two choices. All possible combinations we can obtain are 2^k.
- At time t2 = 2, in the worst case we have 2^k workers or freelances. For each of them the choices we can do are again 2^k, so considering all of them we obtain 2^{2k} possibilities. Now, as we did in the previous exercise, for each group of 2^k possibilities related to a worker or a freelance we compute the minimum cost choices and choose that discarding all the others. In this way, in the following instant we have again 2^k workers or freelance in the worst case.
- This reasoning is valid also for a general instant $t \in (2, ..., T)$.

So this exercise is a generalization of the previous one: before every worker had all the possible skill, so at each instant we only had or a worker or a freelance, that is 2 possible choices at instant 1 and 2^2 at every other instant. Now a worker only has one skill so we could have 2^k worker at instant 1 and 2^{2k} at every other instant.
This leads to a running time that is $O(T * 2^{2k})$ because at each instant we have to consider at most 2^{2k} possibilities.
The correctness proof simply follow from the proof of the previous exercise extended to considering 2^{2k} cases at each instant and not just 2^2.

5 Ex 5

5.1 Ex 5.1

We are given a weigthed graph $G(V, E)$ and an edge $e \in E$.
I have to find an algorithm that check the presence of the edge e in the given graph and this algorithm has to run in $O(|V| + |E|)$ time.
Well, I can exploit the "MST cycle property" of a graph. It says that "For any cycle C in the graph, if the weight of an edge e of C is larger than the weights of all other edges of C, then this edge cannot belong to an MST.

Algorithm 3 Find if a given edge is in an MST of a given Graph

```
procedure IN_MST($G, V, E, ed$)
    $V \leftarrow$ *all distinct instances in D*
    for all $e \in E$ do
        if $weight(e) < weight(ed)$ then
            *setLabel(e, UNEXPLORED)*
        end if
    end for
    for all $v \in V$ do
        *setLabel(v, UNEXPLORED)*
    end for
    $s \leftarrow$ *getFirstEndpoint(ed)*
    $t \leftarrow$ *getSecondEndpoint(ed)*
    $in\_MST \leftarrow$ path_DFS(G, V, E, s, t)
    return $\neg in\_MST$
end procedure
```

Algorithm 4 Find if a given edge is in an MST of a given Graph

```
procedure PATH_DFS($G, V, E, s, t$)
    setLabel(s, VISITED)
    if $s = t$ then
        return TRUE
    end if
    for all $e \in G.incidentVertices(s)$ do
        if $getLabel(e) = UNEXPLORED$ then
            $w \leftarrow opposite(s, e)$
            if $getLabel(w) = UNEXPLORED$ then
                setLabel(e, DISCOVERY)
                $found \leftarrow path\_DFS(G, V, E, w, t)$
                if $found = TRUE$ then
                    return TRUE
                end if
            else
                setLabel(e, BACK)
            end if
        end if
    end for
    return FALSE
end procedure
```

Now I'll explain why this works and how to use MST cycle property: by not considering all the edges that have a weigth equal o more than e, I check

if there still exists a path between u and v, the endpoints of e by computing a path_DFS by starting from u or v.
If in this spanning tree u and v are still connected by a set of edges (of course they will be a subset of E not including e), let's call E', it means that there exists a cycle C made up from E' and e.
Now, since we, computing DFS are considering only edges less-weigthed in respect with e, the given edge will have the maximum weigth in the cycle and this, for the MST cycle property means that it is not included in a MST.
Of course, vice versa, if u and v are not still connected, e will be part of a MST of G.
Finally, the running time of this algorithm is at most the time spent to compute the DFS. This is executed in $O(|V| + |E|)$ time if the graph is represented through list of adjacencies, indeed, in the worst case visit each vertex and each edge.

5.2 Ex 5.2

There are two ways to solve this exercise:

- The first solution could be to use the previous algorithm to first check if exists an MST containing the given edge. Then, if exists compute the MST by using Kruskal's algorithm which runs in $O(|E|log|E|)$ time by using Union-Find structures. Obviously, to be sure to not take another MST which not contains the given edge, I can modify Kruskal by insert the given edge as the first one.

- The second solution is similar but avoid to use the algorithm provided in the previous exercise: indeed, by using Kruskal' algorithm again and still modifying it in this way: when sorting the weigthed edges' set, just make sure that, among all the edges which have the same weigth of the given edge, put the given edge as first. In this way I'm sure of two things:
  - If doesn't exist any MST containing the given edge, then the modified Kruskal' algorithm just will notice it and returns.
  - If an MST containing the given edge does exist, then the one computed from my modified algorithm will surely take the given edge since it is the first among the edges with its same weight.

I just decide to implement the second solution.
To prove its correctness, let's use some property of Kruskal's algorithm and of Union and Find.
Kruskal, start to build the MST considering edges in an increasing order and decide to add an edge in the MST only if its two endpoints don't belong to the same Union-Set.
So there are generally three cases which could happen:

- The given edge simply is not contained in any possible MST of G because it is trivially heavier than every possible necessary MST's edge.
- The given edge is contained in all possible MST and Kruskal takes it.
- The given edge could be part of a possible MST but Kruskal chooses another one.
  First, I've to say that nor A or B are connected to the graph only through e, otherwise it trivially belongs to all possible MST.
  Let's suppose that the given edge e connects the A and B nodes. Now, if exists an MST containing e but Kruskal didn't choose it, it's because there exists an edge connecting A or B to the MST with the same weight of e. Let's see why.
  Now, let's call the other possible edge e' and suppose it is the only other edge that connects A (or B or both it is equal) to the graph. If $e' > e$ then there are three cases: let's suppose e' connects A and C to the MST.
  - If e' is the only incident edge to C, then Kruskal will take it anyway, so both e and e' are taken.
  - If e' is not the only incident edge to C, then or Kruskal will take it anyway, so both e and e' are taken or there exists another less heavier path which connects C to the MST and then also e will be taken.
  - Trivially, if C = B, Kruskal will take the minimum weigthed edge and hence e.

  The same reasoning could be done with done if e' connects B and C.
  Now, if $e' < e$, surely Kruskal already checks if there exists a path from A to B through C that is less heavier then e, but in this case, simply e will not belong to any MST.
  So the only case to choose e' instead e is that e' = e. I handle this case by simply put e first among the other edges with the same weigth such that Kruskal will consider it as the first among that edges.
  By simply iterating this reasonment I can prove the correctnes also if there are other edges which connect A (or B) to the graph.

The changes done on the Kruskal algorithm don't increase its running time when using Union and Find because they simply check two indices and swap the related items. So my algorithm runs in $O(|E|log(|E|))$ due to the edges' sorting.

```
parent = {}
rank = {}

#UNION & FIND definition
def make_set(vertex):
    parent[vertex] = vertex
    rank[vertex] = 0

def find(vertex):
    if parent[vertex] != vertex:
        parent[vertex] = find(parent[vertex])
    return parent[vertex]

def union(vertex1, vertex2):
    root1 = find(vertex1)
    root2 = find(vertex2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
        if rank[root1] == rank[root2]:
            rank[root2] += 1
##########################

def kruskal_alg(graph, e):
    result = False
    minimum_spanning_tree = set()
    for vertex in graph['vertices']:
        make_set(vertex)
    edges = list(graph['edges'])
    edges.sort()
    weights = []
    for edge in edges:
        weights.append(edge[0])

    edges[edges.index(e)] = edges[weights.index(e[0])]
    edges[weights.index(e[0])] = e

    for edge in edges:
        weight, vertex1, vertex2 = edge
        if find(vertex1) != find(vertex2):
            if e == edge:
                result = True
            union(vertex1, vertex2)
            minimum_spanning_tree.add(edge)

    if result:
        print("\nThe MST found is: ")
        return sorted(minimum_spanning_tree)
    else:
        return "It doesn't exist any MST which contains the given
            edge e"

graph = {
'vertices': ['A', 'B', 'C', 'D', 'E', 'F', 'G'],
'edges': set([
(7, 'A', 'B'),
(5, 'A', 'D'),
(7, 'B', 'A'),
(8, 'B', 'C'),
(9, 'B', 'D'),
(7, 'B', 'E'),
(8, 'C', 'B'),
(5, 'C', 'E'),
(5, 'D', 'A'),
(9, 'D', 'B'),
(7, 'D', 'E'),
(6, 'D', 'F'),
(7, 'E', 'B'),
(5, 'E', 'C'),
(15, 'E', 'D'),
(8, 'E', 'F'),
(9, 'E', 'G'),
(6, 'F', 'D'),
(8, 'F', 'E'),
(11, 'F', 'G'),
(9, 'G', 'E'),
(11, 'G', 'F'),
])
}

print(kruskal_alg(graph, (11, 'F', 'G')))
print()
print(kruskal_alg(graph, (7, 'B', 'E')))
```

6 References

1. https://en.wikipedia.org/wiki/Metric_k-center

2. https://en.wikipedia.org/wiki/Bipartite_graph

3. https://en.wikipedia.org/wiki/Kruskal's_algorithm

4. Github's link to ex. 4.1's code: https://github.com/michaelcapponi/AD-Homework1/blob/master/ex4.1.py

5. Github's link to ex. 5.2's code: https://github.com/michaelcapponi/AD-Homework1/blob/master/ex5.2.py