Algorithm Design

Homework 1

Pietro Spadaccino, 1706250

Exercise 1

This problem is a metric k-center problem. There is a known greedy algorithm, as described in [1], which solves this problem with an approximation factor of 2. We will use this algorithm with some modifications and we will show that, in our case, it is optimal.

Algorithm

The algorithm computes the optimal permutation $\Pi(X)$ for every $k \in \{1, \dots, |X|\}$. Initialize the algorithm by iterating on all the distances of the metric space (X, d) in order to find pairs of points $x, y \in X$ such that $d(x, y) = 0$; when such a pair is found, delete y from X and append it to a queue D. By doing so at the end of the initialization we have no pair of points in X having distance equal to 0, we'll explain why. Start by picking a random point x_1. Let's create a queue Π and insert the point x_1 into it. For every point $y \in X$ calculate and store the distance $d(x_1, y)$. Pick the point x_2 that has the maximum distance from x_1, remove it from X and append it to Π. Start another iteration expanding Π with x_3 such that $d(x_2, x_3)$ is maximized, and continue iterating until X is not empty. Now merge the two queues by appending the elements of D to Π.

Running time

The initialization pass requires to iterate on all pairs of points in X, so we have a constant $\mathrm{O}(|X|^2)$. In the worst case we have $|X|$ iterations for choosing the centres, where in the first iteration we have to calculate $|X| - 1$ distances, in the second $|X| - 2$ and so on. Therefore the total cost of calculating the distances is $\mathrm{O}(|X|^2)$. The total cost of the algorithm is then $\mathrm{O}(|X|^2)$.

Proof of correctness

We first show that the greedy algorithm described before has a factor of approximation 2 in the general case, then we explain why in our problem it outputs the optimal solution. We call Opt the maximum distance between a point and its closest center.

Let's first consider the trivial case of $k = |X|$, where the optimal strategy of placing the centers is one for each point (in this case $Opt = 0$). We have that Π is composed by all points of X, $|\Pi| = |X|$, and since each point is a center the solution is correct.

Now let's consider the case where $k < |X|$. By contradiction, assume that the maximum distance $\bar{d}$ between a point $\hat{x}$ and its closest center is strictly greater than two times Opt: $\bar{d} > 2Opt$. This implies that distances between centers are $\geq \bar{d} > 2Opt$, because at every iteration the farthest point is selected as center. Let S be the set composed by k centers and the point $\hat{x}$, so that the distance between every pair of point in S is $> 2Opt$. Since I have k clusters in the optimal solution and $|S| = k+1$, there exists at least one pair in S which is in the same cluster in the optimal solution, i.e. it exists pair of points $x, y \in S$ such that they have the same closest center c and also c is a center in the optimal solution. Since in the optimal solution all points have maximum distance from the closest center of $\leq Opt$, the triangle inequality $d(x, c) + d(y, c) \geq d(x, y)$ is not respected: $d(x, c) + d(y, c) \leq 2Opt$ because x, y are in the same cluster in the optimal solution, but we were assuming that $d(x, c) + d(y, c) > 2Opt$. This is an absurd, therefore the algorithm must have a degree of approximation of 2.

Now we show that in the case, where the admissible distances are only $d \in \{0, 1, 3\}$, this algorithm is optimal. We call the maximum distance of a point and its closest center found by our algorithm Sol, while the optimal one is still called Opt.

- In the case $Opt = 0$, we have that a center must be placed on every distinct point, where by distinct we mean points $x, y \,|\, d(x, y) = 0$. Our algorithm does so: it places the centers on distinct point first, so if $Opt = 0$ also $Sol = 0$.

- In the case where $Opt = 1$, then our solution respects the 2 factor approximation, and it's valid $Sol <= 2Opt$. But having $Sol <= 2$ implies that $Sol = 1$. Also in this case $Sol = Opt$.

- With the same reasoning, if $Opt = 3$ then also $Sol = 3$.

Additional notes

The initialization step is necessary, otherwise the algorithm could return a sub-optimal solution. Let's take as an example a case where $X = \{a, b, c, d\}$ where all pairs of points $x, y \in X$ have distance $d(x, y) = 1$ except for the pair (a, b) which has distance 0. Let's not run the initialization step, which would remove a or b, and start the algorithm by picking a as the first center. Then the algorithm could continue select centers by picking c, because $d(a, c) = 1$, and b, because $d(c, b) = 1$. For $k = 3$ the optimal solution is $Opt = 0$, one center in every distinct point $\{a, c, d\}$, but the solution returned by this run of the algorithm has $Sol = 1$, since point d has distance 1 from its closest center.

Exercise 2

First, we will show how to derive a graph from the input then how to solve it using a max-flow algorithm.

Problem representation

We want to create a bipartite graph $G(V, E)$ such that there exist two disjoint sets of vertices $S \cup A = V$ and $\forall v \in V,\ v \in S \vee v \in A$. We populate the set A by creating a vertex for each avenue and the set S with a vertex for each street. Then we add an edge between two nodes $u \in A, v \in S$ if at the intersection of the avenue represented by u and the street represented by v there is a checkpoint. We note that by adding edges in such a way the graph is indeed bipartite, because a checkpoint could be placed only between a street and an avenue, thus no edge can connect two streets or two avenues.

Algorithm

Before starting, we note that, in the way we have constructed G, our problem is asking to select the minimum number of nodes such that every edge has one endpoint in at least one selected node. So our problem is indeed a vertex cover problem, which isn't hard on bipartite graphs.

What we want to do, is to find a maximum matching and then finding the minimum vertex cover using part of the proof of Kőnig's theorem [2]. This vertex cover will be the solution of our problem.

In order to find a maximum matching we can use max-flow algorithms. First we add source s and sink t nodes, then we connect s to all nodes in S and t to all nodes in A (note that if we inverted s and t the algorithm would work fine as well). Finally we assign to each edge e a unitary capacity $c(e) = 1$. Now we are ready to use the Ford-Fulkerson algorithm finding the max-flow and then retrieving the maximum matching of the graph.

In order to construct a minimum vertex cover, we denote by U the set of unmatched nodes in A, and let's define $Z = U \cup P$ where P denotes the set of vertices that are connected to U by alternating paths (paths that alternate between an edge in the matching and another one not in the matching). Following this construction, we have that $K = (A \setminus Z) \cup (S \cap Z)$ is the minimum vertex cover therefore the solution of our problem.

Running time

Our algorithm uses a call to one of the implementations of Ford-Fulkerson, for example Edmonds-Karp, which runs in $O(|V||E|^2)$ as reported in [3]. Next, the algorithm performs the construction of K: in order to find alternating paths we have to do a DFS for each node in U requiring $O(|V|(|E| + |V|))$ in the worst case. Therefore the running time of the algorithm is $O(|V||E|^2 + |V|(|E| + |V|)) = O(|V||E|^2 + |V|^2)$.

Proof of correctness

We have discussed earlier why our problem is indeed a vertex cover problem because of the way we constructed the graph. Now, in order to show the correctness of the algorithm, we need to prove first that $K = (A \setminus Z) \cup (S \cap Z)$ is a vertex cover, then that it is minimal.

We first split the edges into edges that are part of an alternating path and edges that are not part of an alternating path. Remembering that the graph is bipartite, we can tell that if an edge is part of an alternating path, then it must have one of the endpoints in $S \cap Z$, otherwise it has one in $A \setminus Z$. Since each edge has at least one vertex in K, then K is a vertex cover.

Now we have to prove that K is a minimal vertex cover. First, let be M a maximum matching for G, then $|M|$ is the lower bound for the cardinality of any vertex cover: every vertex can cover at most one edge of the matching M, because otherwise M wouldn't be a matching, and if we had less than $|M|$ vertices then at least one of them must cover at least two edges of M. We have that every vertex in K is an endpoint of one matched edge. Indeed if $v \in (A \setminus Z)$ then v must cover a matched edge because $(A \setminus Z) \subseteq (A \setminus U)$ since $Z \supseteq U$. Also every vertex $v \in (S \cap Z)$ must cover and endpoint of a matched edge: let's consider the alternating path to v and let's assume by contradiction that v doesn't cover any matched edge. This path starts with an unmatched edge from a node $u \in U$ and necessarily ends in v. But then we could take this path and invert the matching of the edges, thus resulting in a matching of cardinality $|M| + 1$, which is an absurd. We have now proven that $|K| = |M|$, thus is a minimal vertex cover. Therefore K is a correct solution for our problem.

Exercise 3

First we describe how we represent the input of the requested problem, then we will show that it is NP-complete.

Problem representation

I can construct a graph $G(V, E)$ where every vertex $v \in V$ is a person and $v \in M \vee v \in F$ holds, with M, F disjoint sets. Between a pair of vertices $u, v \in V$ there is a unweighted edge if $w(u, v) = 1$.

Then, the problem asks me to return a subset of people maximizing $\frac{1}{|I|} \sum_{x,y \in I} w(x, y)$, or, because of how we defined the graph, to return a subgraph of G maximizing its density.

Therefore my final problem is: given a graph $G(V, E)$, where $v \in M \vee v \in F$ with M, F being disjoint sets, find a subgraph $G'(E', V')$ such that $|V' \cap M| = |V' \cap F|$ with $\max\left(\frac{|E'|}{|V'|}\right)$.

Proof

In order to show that the problem is NP-complete I must be able to:

- Verify in polynomial time whether or not a given solution is admissible;
- Show that our problem is at least as hard as another known NP-complete problem.

For the verification, I just have to check if *Sol* satisfies the constraint about having the same number of males and females, and it is easy to do in polynomial time just by counting for each vertex v if $v \in M$ or $v \in F$. If the check is positive, then *Sol* is acceptable. Therefore any solution can be verified in polynomial time and our problem is in NP.

For the second point, I have chosen the clique problem, which tells whether or not there exists a clique of size k in a given graph, as a known NP-complete problem. Let it be problem B and our problem be A. Assuming that we have an algorithm Alg_A capable of solving A, I have to show that there is an algorithm Alg_B that performs the following steps:

- Transforms an instance of problem B into an instance of problem A;
- Uses Alg_A to solve the new instance of A obtaining a solution Sol_A;
- Transforms back the solution Sol_A into a solution Sol_B suitable for problem B.

The running time of the transformation steps should lie below the lower bound of Alg_A. Since we don't know this lower bound, we can assume that if the transformations are done in a polynomial number of steps, then the running time of Alg_B is within the same order of magnitude of Alg_A.

We start by showing how to transform an instance of the clique problem into an instance of our problem. Let $G_B(V_B, E_B)$ and k_B be the inputs of the clique decision problem. I start by constructing a graph $G_A(V_A, E_A)$ where the $V_A = V_B$, $E_A = E_B$ and setting all vertices into males $M = V_B$. Now I create a clique of size k_B, putting all its vertices inside F set and adding it to G_A, which will be the input graph of our problem.

We can now launch Alg_A to solve this new instance, yielding a solution Sol_A. We note that (assuming obviously $|V_B| \geq k_B$, or $|M| \geq |F|$) any Sol_A will have the same number of males and females and this number is equal to k_B: the whole clique made by k_B females is always returned in any Sol_A. To show this, let's consider the case where there are no edges between the vertices representing the males $|(M \times M) \cap E_A| = 0$. In this case, we have that the lower bound of the density of a solution with n males and n females is given by:

$$\frac{\frac{n(n-1)}{2} + 0}{2n} = \frac{n+1}{4}$$

We can see that the density is increasing with the number of selected nodes. Considering that:

- if there were edges between males the density would increase even more;
- the solution of our problem maximizes the density;
- we can't select more than $2k_B$ nodes because of the constraint on the same number of males and females;

then the clique made of k_B females is returned in any solution, alongside some k_B male vertices.

Now we can transform back Sol_A into a solution for problem B. This can be done just by deleting the females from Sol_A and checking whether or not its male vertices, the ones from G_B, form a clique of size k_B. Alternatively we can check if the density of the solution equals the maximum possible density, which can be achieved only in the case where the returned males form a clique. In both cases these transformations can be done easily in poly-time.

The reduction is now finished and we have shown that the problem is NP-complete.

Exercise 4

We will discuss the algorithm, running time and correctness for both 4.1 and 4.2. The code is reported in the appendix.

Algorithm

Before we start, we note that is not optimal to have a hired worker and a freelance at the same time. Indeed, if I had a task at time t and I already have a worker at time t who can accomplish that task, there is no reason to also pick a freelance, because the total cost would be $s + c_t$ instead of only s. Otherwise, if there is no task at time t, I may have a hired worker or not, depending on the optimal strategy, but surely I don't need a freelance, because it wouldn't accomplish any task.

Our algorithm is based on dynamic programming, and it will use an auxiliary table $M(t, hired)$, where t is a time instant and *hired* is a boolean value $\in \{0, 1\}$ being true if at time t we already have a hired worker, false otherwise. The value $M(t, 1)$ represents the minimum cost achievable at time t in case we have already a hired worker, and $M(t, 0)$ the minimum cost achievable at time t in case we don't have any hired worker.

We populate M at time t by recursion on the next time step $t + 1$, evaluating the minimum cost between having a hired worker or a freelance, if there is a task at time t, or between firing a worker or to keep it for future tasks, if there is no task at t. We also have to base our decision on the presence of a worker hired previously. Here is reported the recurrence equation:

$$M(t,h) = \begin{cases} \min(s + M(t+1,1),\ S + c_t + M(t+1,0)) & \text{if } t \in jobs \wedge h = 1 \\ \min(C + s + M(t+1,1),\ c_t + M(t+1,0)) & \text{if } t \in jobs \wedge h = 0 \\ \min(s + M(t+1,1),\ S + M(t+1,0)) & \text{if } t \notin jobs \wedge h = 1 \\ M(t+1,0) & \text{if } t \notin jobs \wedge h = 0 \end{cases}$$

Now that M is populated, we make use use of it for constructing the solution: it is represented by a boolean vector H of size T:

- H_t is true: at time t we must have a hired worker, meaning that if we had a worker at $t - 1$ then keep it, otherwise hire one.
- H_t is false: at time t we must not have a hired worker, fire him if we have one or, if we don't have one, don't hire. In this case, if there is a task at time t, we have to take a freelance.

Note that not all values of M are the value of the final optimal solution: even if $M(\bar{t}, 1) < M(\bar{t}, 0)$ for some $\bar{t}$ it may be suboptimal to hire at time $\bar{t} - 1$, because by doing so we lose information about what happened before $\bar{t} - 1$. For this reason, $M(1, 0)$ is the value of the final optimal solution, because it is the cost at the first time step starting with no hired workers. We have to retrieve the optimal strategy iteratively, trying to reconstruct the cost of $M(1, 0)$ through hiring and firing at each time step t:

- In case I have a worker at t, I keep it if I can reach the optimal cost with a hired worker $M(t + 1, 1)$ by subtracting the salary s from the optimal current cost;
- In case I don't have a worker at t, I hire it if I can reach the optimal cost with a hired worker $M(t+1, 1)$ by subtracting the hiring cost and the salary.

In both cases if I can reach the optimal cost $M(t + 1, 1)$ then I set $H_t = 1$, otherwise $H_t = 0$. By doing so, we can construct the vector H containing the solution of the problem.

For 4.2, we generalize the approach of 4.1. Before we report the pseudocode we introduce some notation:

- $steps = \{1, 2, ..., T\}$;
- $jobs \subseteq steps$ and $|jobs| = n$;
- $W = \{W_{jobs_1}, W_{jobs_2}, ..., W_{jobs_n}\}$;
- $W_i \subseteq \{1, 2, ..., k\}$;
- $hired$ as a boolean vector of size k. The value of $hired_i$ is 1 if I have hired the employee of type i;
- $C_{T,k}$ matrix representing the cost $C_{t,i}$ for outsourcing at time t by worker of type i;
- $M_{T,2^k}$ matrix representing the minimum achievable cost $M(t, hired)$ at time t with already hired workers contained in $hired$. Note that $hired$ is one of the 2^k instances of the vector $hired$, since it is a boolean vector with size k.

The function $find\_opt(t, hired)$ will yield the minimum cost at time t if, at time t, we already have the workers represented inside the $hired$ vector.

```
find_opt(t, hired):
  if t > T:
    return 0

  cost = empty vector

  for every disposition of hired -> disp:
    for i in [1, K]:
      if disp[i] == 1:
        if hired[i] == 1:
          cost[disp] += s
        else:
          cost[disp] += C+s

      if disp[i] == 0:
        if hired[i] == 1:
          cost[disp] += S
        else if t in jobs && i in Wt:
          cost[disp] += C[t][i]

  opt =  min( cost[1]+find_opt(t+1,1), cost[2]+find_opt(t+1,2), ... )
  M[t][hired] = opt
  return opt
```

where:

- using `for every disposition of hired` we mean iterating on all possible values of *hired* vector: given that it is a boolean vector of size k, we have a total of 2^k possible dispositions;
- I can consider a disposition `disp` as a number, thus I can use `cost` as a vector accessible by integer indices.

The function returns the minimum cost of transitioning from `hired` to a certain disposition plus the minimum cost achievable having that same disposition at the next time instant. Once the table is populated, we need to retrieve the solution. For simplicity, I define a subroutine $cost(h1, h2)$ which yields the cost of going from the configuration of hired workers $h1$ to configuration $h2$, where both are boolean vectors like *hired* discussed before.

```
cost(h1, h2):
  c = 0
  for i in [1, K]:
    if h1[i]==1 && h2[i]==1:
      c += s
    if h1[i]==1 && h2[i]==0:
      c += S
    if h1[i]==0 && h2[i]==1:
      c += C+s
  return c
```

With this subroutine, we can construct the solution H in a similar way of what we have done in 4.1, where H is represented as a vector of size T where each element is a vector of size k: $H_{t,i} = 1$ if at time t we need to hire the worker of type i. If $H_{t,i} = 0$ and still $t \in jobs$, then we need to hire a freelancer to execute that task. Below is reported the pseudocode to retrieve the vector H:

```
H = empty list
prev = {0,0,...,0} k times
for t in [1,T]:
  H[t] = argmin( cost(prev, disp)+M[t+1][disp] )
  prev = H[t]
```

where `argmin` returns the disposition `disp` that minimizes the expression over all possible dispositions.

Running time

For 4.1 we have a table M of size $2T$ to be populated, requiring $O(T)$. Next accesses to the table will require constant time, and gathering the final solution is linear respect to T, therefore the time complexity is $O(T)$.

For 4.2 we have that the recursion tree has depth T, and each call performs other new 2^k calls (the number of dispositions discussed earlier), with a cost of $O(2^{kT})$. We can improve this cost by accessing the value inside $M(t, hired)$ instead of calculating it again, if we already have it, but we still pay $O(2^k T)$, because we need to populate M which has size $2^k T$. Then, in order to retrieve the solution we have to scan the entire table, calling the subroutine $cost(h1, h2)$ for each entry, thus having $O(2^k Tk)$. But again, we can improve this cost if we calculate only once all costs for all possible pairs of dispositions and saving the results in an auxiliary table. This result in a overall cost of $O(2^k T + 2^{2k}) = O(2^k(2 + T)) = O(2^k T)$.

Proof of correctness

For 4.1 we must prove by induction that the generated table M is correct: we have to prove that $M(\bar{t}, hired) = OPT_{\bar{t},hired}$ is the minimum cost if at time $\bar{t}$ we already have a hired worker ($hired = 1$) or we haven't ($hired = 0$). As base case we consider $M(T+1, 1) = M(T+1, 0) = 0$, because we don't have to pay any severance cost to hired workers at the end of the time steps. Without loss of generality, we consider the case where there is a task at time $\bar{t}$ and we have already have a hired worker. By construction, $M(\bar{t}, 1) = min(M(\bar{t}+1, 1) + s,\ M(\bar{t}+1, 0) + S + c_t)$. For the inductive hypothesis $M(\bar{t}, 1) = min(OPT_{\bar{t}+1,1} + s,\ OPT_{\bar{t}+1,0} + S + c_t)$. But if i take $M(\bar{t}, 1)$ as the minimum of the only two correct possibilities, then it must be the optimal value $M(\bar{t}, 1) = OPT_{\bar{t},1}$. We can do the same reasoning for $M(\bar{t}, 0)$ and for time steps $\bar{t}$ where there isn't a task.

The solution is then retrieved from M by trying to reconstruct the steps having optimal cost. As discussed earlier, $M(1, 0)$ is the cost of the final optimal solution, because we assume that before the first time step we don't have any hired worker. So the optimal solution is made by actions like hiring, firing, paying salary and outsourcing which will have an overall cost of $M(1, 0)$. If we manage to find those steps whose sum is $M(1, 0)$, we have found an optimal strategy. More formally, every cost of a solution can be expressed as $\alpha s + \beta C + \gamma S + outsourcing$ with α, β, γ integers. In order to find the optimal solution, this cost must be equal to $M(1, 0)$, or in other words $M(1, 0) - (\alpha s + \beta C + \gamma S + outsourcing) = 0$. We can iterate by setting a variable $cost = M(1, 0)$ and at each step t subtracting s, C, S or some outsourcing cost from $cost$, depending on the choosed action. If we arrive at $t = T$ with $cost = 0$ the steps made by the solution have an optimal cost.

For 4.2, we can prove the correctness of M following the same reasoning done for 4.1. As base case I take $M(T+1, disp) = 0$, for every possible disposition $disp$. By construction, $M(t, hired) = cost[disp^*] + M(t+1, disp^*)$ minimizes the cost of transitioning from worker configuration $hired$ to $disp^*$ plus the cost $M(t+1, disp^*)$. But then, $M(t, hired) = cost[disp^*] + OPT_{t+1,disp^*}$. Given that $cost[disp^*]$ is minimized, we have $M(t, hired) = OPT_{t,hired}$. When retrieving the solution, we try to reconstruct the optimal steps, finding at each step t the configuration $disp_t$ that minimizes the cost of transitioning from configuration $disp_{t-1}$ to $disp_t$ plus the optimal cost at the next step having $disp_t$ as configuration.

Exercise 5

We will discuss the algorithm, the running time and the proof for both 5.1 and 5.2. Because 5.2 uses the algorithm from 5.1, I'm going to implement also 5.1.

Algorithm

Let a and b be the endpoints of the edge e.

For 5.1, I perform a DFS on a sub-graph $G'(V', E')$, with $V' = V$ and $E' = \{f \in E \mid w(f) < w(e)\}$. The DFS must be stated from a and has to explore only its connected component. If a and b get connected during the DFS, i.e. a and b are in the same connected component, then e cannot be part of any MST, otherwise there is at least one MST containing it.

Now for 5.2 I've to find a minimum spanning tree containing e. The algorithm starts by calling 5.1 to determine if e is included in some MST. If yes, then proceeds by performing Kruskal's algorithm with a slight modification: when all the edges are sorted in a non-decreasing order inside a list l, if there are multiple edges with the same weight as the one of e, then take e and put it before all other edges with the same weight. We will show that the MST returned by Kruskal contains e.

Running time

5.1 is characterized by a depth-first search on a connected component. In the worst case the graph has a single connected component, so we have a time complexity of $\mathrm{O}(|V| + |E|)$.

5.2 is just a call to 5.1 and another call to Kruskal's algorithm, so it is $O(|E| \log |E|)$.

Proof of correctness

For 5.1, I have to prove the two cases of the algorithm: "*if a and b are in the same connected component of G' then e is not part of any MST*" and "*if a and b are not in the same connected component of G' then there is at least a MST containing e*".

- For the first one it is a simple application of the MST cycle property, which states: "*For any cycle C in the graph, if the weight of an edge e of C is larger than the individual weights of all other edges of C, then this edge cannot belong to an MST*". I consider the cycle in the graph

G composed by the path from a to b found by the DFS plus the edge e. By construction, the path found by the DFS has edges with weight strictly lower than $w(e)$, so we have a cycle C where the weight of e is larger than the individual weight of all the other edges in C, therefore, from the cycle property, there cannot be any MST containing e.

- For the second I can't use the cycle property, because it lacks of the inverse. By construction of G', I have two distinct connected components D and F, such that $a \in D$, $b \in F$. We can consider these connected components as cuts in the original graph G. Note that the cut-set of both includes $n \geq 1$ edges having weight $\geq w(e)$, where n is at least 1 because there is always the edge e between them, and weight $\geq w(e)$ because otherwise both endpoints of the edge would be contained into D or F. To have a spanning tree, obviously all the vertices must be connected, so at least one edge f in the cut-set of D and F must be part of a spanning tree. But we want a minimum spanning tree, thus we want f to have the minimum possible weight, and we have shown that $w(f) \geq w(e)$, so the best way of choosing f is choosing e, or another edge with equivalent weight. Therefore e must be included in at least one MST.

For 5.2 we have to prove that e is in the MST returned by our (slightly) modified Kruskal's algorithm. We start by noting that our modification on Kruskal doesn't impact its correctness, because after we put e as the first of its weight, the list of edges is still in non-decreasing order. The first step of the algorithm is running 5.1 to determine whether or not e can be contained in any MST. If it answers positively, then we are in the situation described in the second point of 5.1, where we have at least two connected components D, F with $a \in D$, $b \in F$ separated by edges of weight $\geq weight(e)$. This means that, by using only edges of weight $< weight(e)$, is not possible to have a spanning tree, nor a minimum spanning tree. Kruskal's algorithm, after examining all the edges with weight $< weight(e)$, must be in this same state, with at least a connected component containing a and another containing b. Now Kruskal starts examining edges with weight $\geq weight(e)$ starting from the lower possible ones, so edges with weight $= weight(e)$. But we are sure that Kurskal is going to select e because it is the first among the edges with equal weight, and it is going to pick it because e is between two distinct connected components. Therefore e is included in the returned MST.

Appendix

Here we list the code for exercises 4 and 5. In the attached files it is reported the same code, in case copy-paste doesn't work. All the code is tested on Python 3.6.

Exercise 4.1

```
from collections import defaultdict

# Params
T = 15
C, S = 5, 20
c = [15] * T
s = 10
M = [[-1,-1] for i in range(T)]
jobs = [0,1,2,2,3,4,6,7,12,14]


def find_opt(t, hired):
  next_hired = M[t+1][1] if M[t+1][1]!=-1 else find_opt(t+1,1)
  next_no_hired = M[t+1][0] if M[t+1][0]!=-1 else find_opt(t+1,0)

  if t in jobs:
      if hired:
          w_hired, wo_hired = s+next_hired, S+c[t]+next_no_hired
      else:
          w_hired, wo_hired = C+s+next_hired, c[t]+next_no_hired

  else:
      if hired:
          w_hired, wo_hired = s+next_hired, S+next_no_hired
      else:
          w_hired, wo_hired = C+s+next_hired, next_no_hired

  M[t][hired] = w_hired if w_hired<wo_hired else wo_hired
  return M[t][hired]


def get_sol():
  H = [-1] * T
  hired_previously = 0 #no workers at t=0
  for t in range(T):
      if hired_previously:
```

```
            H[t] = (M[t][1]-s == M[t+1][1])
        else:
            H[t] = (M[t][0]-C-s == M[t+1][1])
        hired_previously = H[t]
    return H

# Make no job concurrent and update outsourcing cost
def make_jobs_unique():
    global jobs
    d = defaultdict(lambda: 0)
    for job in jobs:
        d[job] += 1
    for job, times in d.items():
        c[job] *= times
    jobs = set(jobs)


make_jobs_unique()
M.append([0, 0]) #aux entry: no additional costs at last time step
find_opt(0, 0)
H = get_sol()


print('Optimal hiring strategy')
print(H)
print('Optimal outsourcing strategy')
print([(t in jobs and not H[t]) for t in range(len(H))])
```

Exercise 5

```
import networkx as nx
from collections import defaultdict

def alg_5_1(G, node, visited):
  # DFS on edges with weight strictly lower than weight(e)
  for edge in G.edges(node):
    if G.edges[edge]['weight'] >= G.edges[(a,b)]['weight']:
        #discard edges with weight>=weight(e)
      continue
    if visited[edge[1]]: #discard edge if its endpoint was already
        visited
      continue
    if edge[1] == b:
      return True
    visited[node] = True
    if alg_5_1(G, edge[1], visited):
      return True
  return False


# Identifiers for endpoints of edge e.
a = 100
b = 101

def alg_5_2(G):
  # Kruskal init - sort edges in a non-decreasing order
  edges_sorted = list(G.edges())
  edges_sorted.sort(key=lambda x: G.edges[x]['weight'])
  edges_sorted = [(e[0],e[1]) if e[0]<e[1] else (e[1],e[0]) for e
      in edges_sorted]

  # Make e the first among other edges with same weight
  edges_sorted.remove((a,b))
  for edge, i in zip(edges_sorted, range(len(edges_sorted))):
    if G.edges[edge]['weight'] == G.edges[(a,b)]['weight']:
      edges_sorted.insert(i, (a,b))
      break

  # Union-find methods
  parent, rank = dict(), dict()
  def find(v):
    return parent[v] if parent[v]==v else find(parent[v])
```

```
  def union(u, v):
    root1 = find(u)
    root2 = find(v)
    if root1 != root2:
      if rank[root1] > rank[root2]:
        parent[root2] = root1
        rank[root1] += rank[root2]
      else:
        parent[root1] = root2
        rank[root2] += rank[root1]
      return True
    return False

  # Kuskal main
  MST = set()
  for v in G.nodes():
    parent[v] = v
    rank[v] = 1
  for edge in edges_sorted:
    if union(*edge):
      MST.add(edge)

  return MST


if __name__=='__main__':
  G = nx.Graph()

  G.add_node(a)
  G.add_node(b)
  G.add_node(2)
  G.add_node(3)
  G.add_node(4)
  G.add_node(5)
  G.add_node(6)
  G.add_node(8)

  G.add_edge(a,b,weight=3)
  G.add_edge(2,4,weight=1)
  G.add_edge(5,6,weight=1)
  G.add_edge(a,3,weight=2)
  G.add_edge(a,5,weight=3)
  G.add_edge(6,b,weight=7)
  G.add_edge(5,4,weight=1)
  G.add_edge(2,3,weight=2)
```

```
G.add_edge(b,4,weight=4)
G.add_edge(4,8,weight=2)
G.add_edge(8,6,weight=7)

visited = defaultdict(lambda: False)

e_in_MST = not alg_5_1(G, a, visited)
print("The edge is "+("in some" if e_in_MST else "not in any")+"
    MST")
if e_in_MST:
  print("MST:", alg_5_2(G))
```

References

[1] Wikipedia, *en.wikipedia.org/wiki/Metric_k-center.*

[2] Wikipedia, *en.wikipedia.org/wiki/Kőnig%27s_theorem_(graph_theory).*

[3] Wikipedia, *en.wikipedia.org/wiki/Edmonds-Karp_algorithm.*