

---

# Algorithm Design First Homework

---

Andrea Fioraldi 1692419

December 1, 2018

## EXERCISE 1

This problem can be viewed as a simplification of the K-center clustering problem. This problem has an approximate solution, a greedy 2-approximation algorithm. We used this simplification and we also proved that this approach is optimal for our problem.

$C$  is the subset of  $X$  that contains the centers. The algorithm is the following:

1. Pick an arbitrary point  $c_1$  and insert in into  $C$ ;
2. Pick the point  $x$  with the highest distance from the nearest center  $c_i$ ;
3. Insert  $x$  into  $C$ ;
4. Continue from 1 until  $|C| < k$ ;

In pseudocode:

```

1 algorithm  $k\_centers(G, k)$ 
2    $c := random\_select(G.V)$ 
3    $G.V = G.V \setminus \{c\}$ 
4    $C := \{c\}$ 
5    $distances := \text{new Array}(|G.V|)$ 
6    $distances_i = +\infty \ \forall i = 1 \dots |G.V|$ 
7   while  $|C| < k$ 
8      $new\_c := none$ 
9      $max\_dist := -\infty$ 
10    foreach  $v \setminus \text{in } G.V$ 
```

```

11          $d := \min(\text{distances}_v, G.\text{weight}(c, v))$ 
12         if  $\text{max\_dist} < d$ 
13              $\text{max\_dist} = d$ 
14              $\text{new\_c} = c$ 
15          $c = \text{new\_c}$ 
16          $G.V = G.V \setminus \{c\}$ 
17          $C = C \cup \{c\}$ 
18     return  $C$ 

```

The **distances** list is an optimization used to avoid to compute the distance between a node  $x \in X$  and all centers in all of the iterations. **distances** keeps track of the distance each node from its nearest center. Each iteration only a center  $c_i$  is added, so the distance from the nearest center can remain the same or be updated to  $d(x, c_i)$ .

So this approximation cost is  $O(n * k)$  because in each iteration all nodes are processed and there are  $k$  iterations.

To compute the permutation requested by the exercise we run the algorithm with  $k = n$ , where  $n = |X|$ . This is possible thanks to the fact that in this greedy algorithm the iteration  $i$  is not dependent from the presence of an iteration  $i + 1$  so computing  $k\_center(G, k)$  is equivalent to  $\text{slice}(k\_center(G, n), 1, k)$  (compute using  $k = n$  and after take only the first  $k$  elements of the output).

This greedy algorithm is optimal for our problem thanks to the fact that the distances can be only 0, 1 or 3.

For the K-center this algorithm is a 2-approximation, so the approximated minimized maximum of the distances of each point in  $X$  to the closest center is at most the twice of the optimal one. We will notate the approximated with  $r_A$  and the optimal with  $r_O$ . Regards our problem both  $r_A$  and  $r_O$  can be 0 only if  $|C| = |X|$ , so in this case they are equal. There are other 4 cases to consider:

- $r_A = 1 \wedge r_O = 3$  is not possible because an approximation cannot perform better than the optimal algorithm <sup>1</sup>.
- $r_A = 3 \wedge r_O = 1$  is not possible because  $r_A > 2 * r_O$ ;
- $r_A = 3 \wedge r_O = 3$  is optimal;
- $r_A = 1 \wedge r_O = 1$  is optimal;

Follows that in our problem  $r_A$  must be equal to  $r_O$ , so our algorithm is optimal and not an approximation.

## EXERCISE 2

The problem can be represented with a bipartite graph  $G(L \cup R, E)$ . The nodes of the class  $L$  represents the avenues and the nodes of the class  $R$  the streets. The edges in  $E$

---

<sup>1</sup>A p-approximation  $f$ , as described in [1], perform  $OPT \leq f \leq p * OPT$

are the checkpoints. Using a reversed perspective, the adjacency matrix of such graph is the grid of avenues and streets with the edges represented by checkpoints.

A vertex cover of such graph (the set of nodes so that each edge has at least one endpoint in the set) represents a set of streets and avenues that can cover all checkpoints. A minimum vertex cover is the set of streets and avenues that solves our problem.

Generally, a minimum vertex cover is a hard problem but with bipartite graphs, we can solve it in polynomial time thanks to the König's theorem.

According to [2], the statement is the following:

*In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.*

Its proof offers a method to retrieve the minimum vertex cover from the maximum bipartite matching.

Adding a source node  $s$  to one side and a sink node  $t$  to the other we create a flow network based on the bipartite graph:

$G'(L \cup R \cup \{s, t\}, c, s, t)$

All edges capacities are 0 except the following:

- $c(s, l) = 1 \ \forall l \in L$ ;
- $c(r, t) = 1 \ \forall r \in R$ ;
- $c(e) = \infty \ \forall e \in E$ ;

The amount of flow for each edge can be 0 or 1. For each flow the subset of  $E$  with  $f(e) = 1$  is a matching of cardinality equal to the value of the flow. So a maxflow value is equal to the cardinality of a maximum matching  $M$ . Let  $(S, T)$  the mincut associated with the computed maxflow using Ford-Fulkerson.

Consider the set  $C = (L \cap T) \cup (R \cap S)$ . We want to prove that  $C$  is a vertex cover and  $|C| = |M|$ .

Firstly, assume that  $C$  is not a vertex cover. So we have an edge  $e \in E$  with endpoint in  $L \cap S$  and  $R \cap T$  with capacity  $c(e) = \infty$  (the capacity is always  $\infty$  in the original edges). This is absurd because an infinite capacity edge cannot be in a minimum cutset (it has one endpoint in  $S$  and the other in  $T$ ), so all edges have endpoints in  $(L \cap T) \cup (R \cap S) = C$  and  $C$  is a vertex cover.

Secondly, we have that:

- thanks to the maxflow mincut theorem the cardinality of  $M$  is equal to the capacity of  $(S, T)$ ;
- the capacity of  $(S, T)$  is equal to the number of edges in the cutset that is composed only by edges from  $s$  to  $T$  and from  $S$  to  $t$  (otherwise they would have an infinite capacity);
- such edges are associated to  $(L \cap T)$  and  $(R \cap S)$ , so  $|M| = \text{cap}(S, T) = |(L \cap T)| + |(R \cap S)| = |C|$ .

So a minimum vertex cover can be computed using the Ford-Fulkerson method and is  $(L \cap T) \cup (R \cap S)$ .

We propose a solution that makes use of a variant of Ford-Fulkerson that guarantee termination, the Edmonds-Karp algorithm, which has cost  $O(|V| * |E|^2)$ , and then of two DFS in the residual graph in order to compute  $L \cap T$  and  $R \cap S$ . So the final cost is  $O(|V| * |E|^2 + 2 * |V| + 2 * |E|) = O(|V| * |E|^2)$ .

### EXERCISE 3

We represents our problem using a undirected unweighed graph. Each node of the graph is a friend and an edge  $(v, u)$  is present if  $w(v, u) = 1$ . The goal is to find the subset of nodes  $I$  that maximize  $\frac{1}{|I|} \sum_{x, y \in I} w(x, y)$  respecting the constraint  $|I \cap M| = |I \cap F|$ . In other words, it is the densest subgraph that has an equal number of M and F nodes.

#### CERTIFIER

In order to verify an input  $s$  we create an efficient certifier  $CERT(s, t)$ :

- if  $s \cap M \neq s \cap F$  return no;
- let  $d_s = \frac{1}{|s|} \sum_{x, y \in s} w(x, y)$ ;
- let  $d_t = \frac{1}{|t|} \sum_{x, y \in t} w(x, y)$ ;
- return yes if  $d_s \geq d_t$  else return no;

$CERT$  runs in polynomial time, precisely  $O(|s| + |t|)$ . This certifier can be used in a brute-force over all possibles set of friends  $t$  with  $|t| \leq |s|$ . If all attempts returns yes,  $s$  is the solution.

#### REDUCTION

Given our problem  $X$  and the clique decision problem  $Y$ , we will prove that  $Y \leq_p X$ . The clique decision problem takes as input an undirected unweighted graph  $G(V, E)$  and a number  $k$  and it says yes if the graph contains a clique of  $k$  nodes.  $Y$  is NP-complete. To transform an input for  $Y$  to an input for  $X$  we create a graph  $G'(M, F, E')$  copying the nodes of graph  $G$  into  $M$  and the edges into  $E'$ . Then we create  $k$  nodes in  $F$  and we connect them in a clique adding edges to  $E'$ .

Note that an output of  $X_{ALG}$  contains a clique of F-nodes because a subgraph of a clique is a clique. So if the output contains  $t$  M-nodes then it contains also a clique of F-nodes of size  $t$ .

Assume that exists a clique of size  $k$  composed by  $M$ -nodes and that  $X_{ALG}$  returns an output with less  $M$ -nodes than  $k$  or with  $k$   $M$ -nodes but that is not a clique. This is not possible because the densest graph of  $k$  nodes is a clique by definition and a clique of  $k$

nodes is denser than every clique with fewer nodes (the density of a clique  $\frac{x(x-1)/2}{x}$  is a monotonous growing succession).

Assume that exists a clique of size  $k$  composed by nodes of  $M$  and that  $X_{ALG}$  returns an output with more  $M$ -nodes than  $k$ . This is not possible because we have at most  $k$   $F$ -nodes and so it violates  $|I \cap M| = |I \cap F|$ .

Assume that exists a clique of size  $k$  composed by nodes of  $M$  and that  $X_{ALG}$  returns it as part of the output. Deleting the  $F$ -nodes clique from the output give us the  $M$ -nodes clique. Another result is not possible because it implies that there are edges from  $M$  to  $F$  and this violates our construction of the graph.

So with  $X_{ALG}$ , our black box solver, we can solve an instance of  $Y$  checking if the  $M$ -nodes in the output forms a clique of size  $k$ .

## EXERCISE 4

The related code is in the Appendix.

### PART 1

Assumptions:

- at the beginning we do not have a hired worker;
- at the end if we have and hired worker we do not have to outsource him;

We never use a freelance when we have a hired worker because we must pay a hired worker also when not working, so if we have a hired worker is always convenient to use him.

We denote as  $OPT(x, h)$  the minimum cost of execution of the task  $j_t$  with  $t \in \{x, \dots, T\}$  and with  $h = true$  when we have a hired worker at disposition. So the solution that we are looking for is  $OPT(0, false)$ .

The cost of a single job can be explained using two cases:

When  $h = true$  we must take one of this two subcases, the one with the minimum cost:

- We use the hired worker that we have to pay him  $s$  for the job  $j_t$ ;
- We outsource him paying  $S$  and then we use a freelance paying the cost  $c_t$  for the job  $j_t$ ;

On the contrary, with  $h = false$ , we must consider the subcase of the minimum cost from the following two:

- We assume a hired work paying  $C$  and then we pay  $s$  for the job  $j_t$ ;
- We use again a freelance paying the cost  $c_t$  for the job  $j_t$ ;

We have to not consider the case in which there is not a job at the time instance  $t$  and instead we can simply set the cost  $c_t$  of a freelance to 0 in that case. To prove this claim assume that we do not have a job  $j_t$  at the instant  $t$  and we have a hired worker. In this case, we must choose one of the following possibilities:

- Outsource the hired worker and wait until the next job;
- Pay him to do nothing;

Otherwise, when we do not have a hired worker we do nothing.

Now if we consider a freelance with cost 0 we can express the cost of the "not job" as follows:

When  $h = \text{true}$  we must take one of this two subcases, the one with the minimum cost:

- We use the hired worker that we have to pay him  $s$  for the "not job"  $j_t$ ;
- We outsource him paying  $S$  and then we use a freelance paying 0 for the job  $j_t$ ;

On the contrary, with  $h = \text{false}$ , we must consider the subcase of the minimum cost from the following two:

- We assume a hired work paying  $C$  and then we pay  $s$  for the job  $j_t$  (of course this can never be the minimum);
- We use again a freelance paying the cost 0 for the job  $j_t$  (this is the always chosen option);

So considering a "not job" is equivalent of the previously described situation and we do not have to distinguish the cases in our algorithm, we just set  $c_t = 0$  when there is not a job  $j_t$ .

Returning to  $OPT(x, h)$ , to the single cost of  $j_t$  we must add the cost of the successive jobs until the time instant  $T$ , so we explore all the possible following cases:

$$OPT(x, h) = \begin{cases} \min(s + OPT(x + 1, \text{true}), S + c_t + OPT(x + 1, \text{false})) & \text{if } h = \text{true} \\ \min(C + s + OPT(x + 1, \text{true}), c_t + OPT(x + 1, \text{false})) & \text{if } h = \text{false} \end{cases}$$

This can be traduced into an algorithm that explores all the possible configurations and so a brute-force. In order to gain a polynomial cost, we used the memorization of the results of  $OPT(x, h)$ . We can store the solutions in a matrix  $M_{T+1,2}$  storing in the first column the values of  $OPT(x, \text{false})$  and in the second  $OPT(x, \text{true})$ . Obviously the rows are related to the  $x$ , so encoding  $\text{false} = 0$  and  $\text{true} = 1$  we can get the return value of  $OPT(x, h)$  with  $((M)_x)_h$ . Note that the last row of  $M$  is present in order to avoid to check if we are in the case  $t = T - 1$ .

The algorithm, `min_cost`, is the following:

```

1 algorithm min_cost_aux( $t, T, s, C, S, c, h, M$ )
2   if  $t$  is  $T$ 
3     return 0
4   if  $((M)_{t+1})_1$  is empty
5      $((M)_{t+1})_1 = \text{min\_cost\_aux}(t + 1, T, s, C, S, c, 1, M)$ 
6   if  $((M)_{t+1})_0$  is empty
7      $((M)_{t+1})_0 = \text{min\_cost\_aux}(t + 1, T, s, C, S, c, 0, M)$ 
8   if  $h$  is 1
9     return  $\text{min}(s + ((M)_{t+1})_1, S + c_t + ((M)_{t+1})_0)$ 
10  else if  $h$  is 0
11    return  $\text{min}(C + s + ((M)_{t+1})_1, c_t + ((M)_{t+1})_0)$ 

1 algorithm min_cost( $j, T, s, C, S, c$ )
2    $M := \text{new Matrix}(T + 1, 2)$ 
3   for  $t = 0$  to  $T$ 
4     if not  $j_t$ 
5        $c_t = 0$ 
6   return min_cost_aux(0,  $T, s, C, S, c, 0, M$ )

```

In general the execution of the body of `min_cost_aux` takes  $O(1)$  excluding the recursive calls that it generates. We can count the number of recursion counting the entries of  $M$  that are not empty.

Every time the procedure invokes the recurrence it fills two entries of the matrix. An entry cannot be filled more than one time. The entries in  $M$  are  $2 * T + 2$  and so the algorithm perform at most  $2 * T + 2$  recursions and so the cost is linear,  $O(T)$ .

The proof of correctness is the following:

In the base case  $t = T - 1$ , we consider two options in order to choose the best worker:

- $h = \text{true}$ : the minimum cost is  $\text{min}(s, S + c_t)$ ;
- $h = \text{false}$ : the minimum cost is  $\text{min}(C + s, c_t)$ ;

Now assume that `min_cost` returns the minimum for  $t + 1$  when using a hired worker,  $X$ , and the minimum cost for  $t + 1$  when using a freelance,  $Y$ .

For the case  $t$  we consider two options:

- $h = \text{true}$ : the minimum cost is  $\text{min}(s + X, S + c_t + Y)$ ;
- $h = \text{false}$ : the minimum cost is  $\text{min}(C + s + X, c_t + Y)$ ;

## PART 2

We can covert the works  $j_t$  with  $W_t \subset W$  to  $W_t = W$  using the same consideration for the "no job" of the previous part. So we set  $c_t^j = 0$  when  $j \notin W_t$ .

With this consideration, we can formulate the problem as derived from 4.1. In fact, for each type of worker  $w \in W$  we can just run the 4.1 algorithm and sum all the minimum costs.

In pseudocode (we directly associate a job  $j_t$  with the set  $W_t$  to avoid ambiguity with  $W$ ):

```

1 algorithm min_cost_multi_types( $j, W, T, s, C, S, c$ )
2    $r := 0$ 
3   for  $t = 0$  to  $T$ 
4     foreach  $w \in W$ 
5       if not  $(j_t)_w$ 
6          $c^w_t = 0$ 
7     foreach  $w \in W$ 
8        $M := \mathbf{new} \text{ Matrix}(T + 1, 2)$ 
9        $r = r + \text{min\_cost\_aux}(0, T, s, C, S, c^w, 0, M)$ 
10  return  $r$ 

```

The cost of the loop at line 4 is  $k$ , the cardinality of  $|W|$ . The outer for at line 3 has  $T$  iterations, so the total cost is  $O(T * k)$ . The cost of `min_cost_aux` is  $O(T)$  and it is inside a loop of  $k$  iterations, so the cost of this second loop at line 7 is  $O(k * T)$ . This implies that the total cost of `min_cost_multi_types` is  $O(T * k) = O(|j| * |W|)$ , so it is polynomial in function of the size of the two sets in the input. Considering the number  $L = 2^k$  we can express the cost as  $O(\log_2(L) * T)$ .

To prove that the algorithm is correct we proceed by induction.

Base case is  $k = 1$ : In that case `min_cost_multi_types` is exactly `min_cost`, so it is correct. Assume now that it is correct for the case  $k = i$ . In the case with  $k = i + 1$  we can see that we are adding a new type of worker to  $W$ . The types of workers are independent of each other, and adding the result of `min_cost` using the type of worker  $i + 1$  to the total cost computed in the case  $k = i$  we get the minimum cost.

## EXERCISE 5

The related code is in the Appendix.

### PART 1

Given a weighted graph  $G(V, E)$ , to decide if exists an MST containing an edge  $e \in E$  we can exploit two properties.

The cycle property [3]:

*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.*

The cut property [4]:

*For any cut  $C$  of the graph, if the weight of an edge  $e$  in the cut-set of  $C$  is strictly smaller than the weights of all other edges of the cut-set of  $C$ , then this edge belongs to all MSTs of the graph.*



Without loss of generality, we can also assume that all weights in the graph are distinct because we can always differentiate two equal weighted edges adding a small constant  $c$  to one of the weights without changing the result of the Kruskal algorithm.

The idea is to determinate if starting from one endpoint of  $e$  ( $v$ ) we can reach the other ( $u$ ) considering only the edges with weights lower than the weight of  $e$  ( $w$ ). We call the graph with only these edges  $G'(V', E')$ . If that happens, we have a connected component that contains both  $v$  and  $u$  and so adding  $e$  to such subgraph we obtain a cycle in which  $e$  is the edge with the maximum weight. This violates the cycle property, so  $e$  does not belong to any MST. In the other case, when  $v$  and  $u$  are not connected with edges with weight less than  $w$ , if exists a set  $S$  for which  $v \in S \wedge u \in V' \setminus S$  the cut property implies that  $e$  is in all MSTs. We choose  $S$  as all nodes that can be reached by  $v$  in  $G'$  so that cannot exists an edge with weight less than  $w$  and one endpoint in  $S$  and the other in  $V' \setminus S$ . If such edge exists, the endpoint in  $V' \setminus S$  is reachable from  $v$  so it is a contradiction. This implies that  $e$  is the edge with minimum cost with an endpoint in  $S$  and the other in  $V' \setminus S$ , so  $e$  belongs to all MSTs.

We designed an algorithm, `edge_is_in_mst`, based on DFS in order to verify if exists a path formed by edges with weight less than  $w$  that connects  $v$  to  $u$ :

```

1 algorithm edge_is_in_mst_aux( $G, e, v$ )
2    $r := true$ 
3    $v.visited = true$ 
4   foreach  $i \in G.neighbors(v)$ 
5     if  $G.weight(v, i) \geq G.weight(e)$ 
6       skip
7     if  $i$  is  $e.u$ 
8       return  $false$ 
9     if not  $i.visited$ 
10       $r = \text{randedge\_is\_in\_mst\_aux}(G, e, i)$ 
11 return  $r$ 

```

```

1 algorithm edge_is_in_mst( $G, e$ )
2   return edge_is_in_mst_aux( $G, e, e.v$ )

```

Checking the weight of adjacent edges does not add cost to the DFS since is a  $O(1)$  operation. Also stopping the algorithm when  $e.u$  is encountered does not add any cost, so the final cost of `edge_is_in_mst` is the cost of a DFS,  $O(|V| + |E|)$ .

## PART 2

Given a weighted graph  $G(V, E)$ , to compute an MST that contains a determinate edge  $e$  we designed an algorithm based on Kruskal with a sorted list as a source of edges. This Kruskal version has a cost of  $O(|E| \log |E|)$  because it uses a sorting algorithm based on comparison <sup>2</sup>. The action of removing the edges with the minimum weight is done in constant time.

---

<sup>2</sup>In the implementation we used the standard python sort function that is based on Timsort [5]

Our algorithm is the following:

1. check if  $e$  can be in a MST using `edge_is_in_mst`, if not exit with an error;
2. sort the edges using the weights and store them in a list  $l$ ;
3. extract the edge  $e$  from  $l$ ;
4. add  $e$  to the MST;
5. continue with the iterations of Kruskal using  $l$  as source of edges;

In pseudocode:

```

1 algorithm mst_from_edge( $G, e$ )
2   if not edge_is_in_mst( $G, e$ )
3     error()
4   foreach  $v \in G.V$ 
5     make_set( $v$ )
6   ordered := sort_by_increasing_weight( $G.E$ )
7   ordered = ordered \  $e$ 
8   mst := { $e$ }
9   foreach  $o \in \textit{ordered}$ 
10    if find_set( $o.v$ ) is not find_set( $o.u$ )
11      mst = mst  $\cup$  { $o$ }
12      union( $o.v, o.u$ )
13  return mst

```

The action 1 costs  $O(|V| + |E|)$  as described before. The action 2, the sorting, costs  $O(|E|\log|E|)$ . Action 3 and 4 are in constant time. So the exact cost of `mst_from_edge` is  $O(|V| + |E| + |E|\log|E|)$ . With the assumption of  $|E| > |V|$  (true if  $G$  is connected) our algorithm cost is  $O(|E|\log|E|)$ .

To prove the correctness of `mst_from_edge` we assume that the cut property is verified for  $e$  (`edge_is_in_mst` check for this).

The output  $Y$  is a spanning tree because:

- cannot have a cycle thanks to the check in the algorithm;
- all nodes of  $G$  belongs to  $Y$ ;
- cannot be disconnected, since the first encountered edge that joins two components of  $Y$  would have been added;

For the minimality we proceed by induction proving the proposition  $P$ :

*If  $F$  is the set of edges chosen at any stage of the algorithm, then there is some minimum spanning tree that contains  $F$ .*

- At the beginning, when  $F = \{e\}$ ,  $P$  is correct thanks to the cut property.
- Assume  $p$  is true for some set  $F$  and  $T$  be the MST that contains  $F$ :
  - after choosing the next edge  $g$ , if  $g$  is in  $T$  then  $P$  is verified for  $F \cup \{g\}$ ;
  - else if  $g$  is not in  $T$  then  $T \cup \{g\}$  has a cycle  $C$  and there is an edge  $f$  that belongs to  $C$  and  $T$  but not to  $F$ . Then  $T \setminus \{f\} \cup \{g\}$  is a tree with weight  $\leq$  of the weight of  $T$  because the weight of  $f$  cannot be  $<$  of the weight of  $g$  (or the algorithm would have chosen  $f$  and not  $g$ ).  $P$  is verified for  $T \setminus \{f\} \cup \{g\}$  that is a MST that contains  $F \cup \{g\}$ ;
- by induction  $P$  holds also when  $F$  is a spanning tree (when  $Y$  is  $F$ ) and so it is a MST.

We proved that  $Y$  is a spanning tree, that is minimum and that contains  $e$ .

## APPENDIX

### EXERCISE 4 CODE

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define AT(mat, x, y) (mat)[(x)*2 + (y)]
6  #define MIN(a, b) ((a) <= (b) ? (a) : (b))
7
8  #define HIRED 1
9  #define FREELANCE 0
10
11 int min_cost_aux(int last_worker_type, int jt, int T, int s,
    ↪ int C, int S, int* freelances, char* works, int* matrix
    ↪ )
12 {
13     if(jt == T)
14         return 0;
15
16     int ct = works[jt] ? freelances[jt] : 0;
17
18     if(AT(matrix, jt+1, HIRED) == -1)
19         AT(matrix, jt+1, HIRED) = min_cost_aux(HIRED, jt+1, T, s,
    ↪ C, S, freelances, works, matrix, sol);
20     if(AT(matrix, jt+1, FREELANCE) == -1)
21         AT(matrix, jt+1, FREELANCE) = min_cost_aux(FREELANCE, jt
    ↪ +1, T, s, C, S, freelances, works, matrix, sol);
22
23     int x,y;
24     if(last_worker_type == HIRED) {
25         x = s + AT(matrix, jt+1, HIRED);
26         y = S + ct + AT(matrix, jt+1, FREELANCE);
27     }
28     else {
29         x = C + s + AT(matrix, jt+1, HIRED);
30         y = ct + AT(matrix, jt+1, FREELANCE);
31     }
32
33     return MIN(x, y);
34 }
35
36 int min_cost(int T, int s, int C, int S, int* freelances,
```

```

    ↪ char* works)
37 {
38     int* matrix = calloc(sizeof(int), (2*T +2));
39     memset(matrix, -1, sizeof(int)*2*T);
40
41     int r = min_cost_aux(FREELANCE, 0, T, s, C, S, freelances,
    ↪ works, matrix, sol);
42
43     free(matrix);
44     return r;
45 }
46
47 int main()
48 {
49     int freelances[16] =
    ↪ {1,4,3,2,8,11,7,6,9,11,3,13,5,1,9,100}; //costs ct
50     char works[16] = {1,1,0,1,1,1,1,0,1,1,1,1,0,1,1,1}; //0 if
    ↪ there is not a task at instant t
51
52     int cost = min_cost(16, 4, 6, 3, freelances, works);
53
54     return 0;
55 }

```

## EXERCISE 5 CODE

```

1  import networkx as nx
2
3  def is_in_mst(G, e):
4      visited = [False]*(len(G.nodes()))
5      e_w = G.edges[e]["weight"]
6
7      def aux(v):
8          visited[v] = True
9          r = True
10
11         for i in G.neighbors(v):
12             if G.edges()[v,i]["weight"] >= e_w:
13                 continue
14             if i == e[1]:
15                 return False
16             if visited[i] == False:
17                 r = r and aux(i)
18
19         return r
20
21     return aux(e[0])
22
23
24 def kruskal(G, e=None):
25     # e!=None forces to include edge e in the MST
26     # otherwise standard kruskal is performed
27
28     def find(parent, i):
29         if parent[i] == i:
30             return i
31         return find(parent, parent[i])
32
33     def union(parent, order, x, y):
34         rx = find(parent, x)
35         ry = find(parent, y)
36         if order[rx] < order[ry]:
37             parent[rx] = ry
38         elif order[rx] > order[ry]:
39             parent[ry] = rx
40         else :
41             parent[ry] = rx
42             order[rx] += 1

```

```

43
44     mst = []
45
46     i = 0
47     j = 0
48
49     s_edges = sorted(G.edges(), key=lambda x: G.edges[x]['
        ↳ weight'])
50     if e is not None:
51         s_edges.remove(e)
52         s_edges = [e] + s_edges
53
54     parent = []
55     rank = []
56
57     for n in G.nodes():
58         parent.append(n)
59         rank.append(0)
60
61     while j < len(G.nodes())-1:
62         if i == len(s_edges): break
63
64         u,v = s_edges[i]
65         w = G.edges[u,v]['weight']
66
67         i += 1
68         x = find(parent, u)
69         y = find(parent, v)
70
71         if x != y:
72             j += 1
73             mst.append((u,v,w))
74             union(parent, rank, x, y)
75
76     return mst
77
78 def mst_from_edge(G, e):
79     if not is_in_mst(G, e):
80         raise Exception("edge %s cannot be in a MST" % str(e)
            ↳ )
81     return kruskal(G, e)

```

## REFERENCES

- [1] “Approximation algorithm - Wikipedia.” [https://en.wikipedia.org/wiki/Approximation\\_algorithm#Performance\\_guarantees](https://en.wikipedia.org/wiki/Approximation_algorithm#Performance_guarantees). Accessed: 2018-11-20.
- [2] “Kőnig’s theorem (graph theory) - Wikipedia.” [https://en.wikipedia.org/wiki/K%C5%91nig%27s\\_theorem\\_\(graph\\_theory\)](https://en.wikipedia.org/wiki/K%C5%91nig%27s_theorem_(graph_theory)). Accessed: 2018-11-20.
- [3] “Minimum spanning tree, Cycle Property - Wikipedia.” [https://en.wikipedia.org/wiki/Minimum\\_spanning\\_tree#Cycle\\_property](https://en.wikipedia.org/wiki/Minimum_spanning_tree#Cycle_property). Accessed: 2018-11-14.
- [4] “Minimum spanning tree, Cut Property - Wikipedia.” [https://en.wikipedia.org/wiki/Minimum\\_spanning\\_tree#Cut\\_property](https://en.wikipedia.org/wiki/Minimum_spanning_tree#Cut_property). Accessed: 2018-11-14.
- [5] “Timsort - Wikipedia.” <https://en.wikipedia.org/wiki/Timsort>. Accessed: 2018-11-14.