Sapienza University of Rome

Algorithm Design First Homework

Andrea Fioraldi 1692419

December 5, 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 2 until $|C| < k$;

In pseudocode:

algorithm $k\_centers(G, k)$
    $c := random\_select(G.V)$
    $G.V = G.V \setminus \{c\}$
    $C := \{c\}$
    $distances :=$ **new** $Array(|G.V|)$
    $distances_i = +\infty \ \forall i = 0...|G.V|$
    while $|C| < k$
        $new\_c := none$
        $max\_dist := -\infty$
        foreach $v \in G.V$

```
            d := min(distances_v, G.weigth(c, v))
            if max_dist < d
                max_dist = d
                new_c = c
        c = new_c
        G.V = G.V \ {c}
        C = C ∪ {c}
    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 of each node from its nearest center. In each iteration only a center c_i is added, so the distance from the nearest center can remain the same or be updated to $d(x, c_i)$.
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 on the presence of an iteration $i + 1$ so computing $k\_center(G, k)$ is equivalent to $slice(k\_center(G, n), 1, k)$ (compute using $k = n$ and after take only the first k elements of the output).
Now, we prove that 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 [1].
- $r_A = 3 \wedge r_O = 1$ 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

[1] 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 Kőnig'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 ∞ 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 associted to $(L \cap T)$ and $(R \cap S)$, so $|M| = cap(S, T) = |(L \cap T)| + |(R \cap S)| = |C|$.

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 unweigthed 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 $\frac{1}{|I|}\Sigma_{x,y \in I} w(x, y)$ 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 efficent certifier $CERT(s, t)$:

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

$CERT$ runs in polynomial time, precisely $O(|s| + |t|)$. This certifer can be used in a brute-force over all possibles set of friends t with $|t| \leq |s|$. If all attempts returs 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 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 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 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 fire him;

We never use a freelance when we have a hired worker because we must pay a hired worker also when is not working, so if we have a hired worker is always convenient to use him.
In addition, it is never convenient to use a freelance if we decide to use a hired worker because he can process all the works at time t. So the choice is between using a hired worker or use k freelances when we have k works at time t. This is equivalent to consider the cost of using the k freelances to the cost of using a freelance with a cost equal to the sum of all costs of the k freelances and then consider only a work for each time instant. From now on we consider that each time instant can have 0 or 1 tasks exploiting the previous consideration.
We denote as $OPT(x, h)$ the minimum cost of execution of the task j_t with $t \in \{x, ..., T\}$ 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 task 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 task j_t;
- We fire him paying S and then we use a freelance paying the cost c_t for the task 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 task j_t;
- We use again a freelance paying the cost c_t for the task j_t;

We have to not consider the case in which there is not a task at the time instant 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 task j_t at the instant t (we call it a "no task" time instant) and we have a hired worker. In this case, we must choose one of the following possibilities:

- Fire the hired worker and wait until the next task;
- 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 "no task" as follows:
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 "not task" j_t;
- We fire him paying S and then we use a freelance paying 0 for the task 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 task j_t (of course this can never be the minimum);
- We use again a freelance paying the cost 0 for the task j_t (this is the always chosen option);

So considering a "no task" 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 task j_t.
Returning to $OPT(x, h)$, to the single cost of j_t we must add the cost of the successive tasks until the time instant T, so we explore all the possible following cases:

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

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 $OPT(x, false)$ and in the second $OPT(x, true)$. Obviously the rows are related to the x, so encoding $false = 0$ and $true = 1$ we can get the return

value of $OPT(x, h)$ with $\left((M)_x\right)_h$. 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 (tasks is an array of tuples (time instant, outsourcing cost), T is a number and we must consider as input the time interval $I = \{0...T\}$):

```
algorithm min_cost_aux(t, T, s, C, S, c, h, M)
    if t is T
        return 0
    if ((M)_{t+1})_1 is empty
        ((M)_{t+1})_1 = min_cost_aux(t + 1, T, s, C, S, c, 1, M)
    if ((M)_{t+1})_0 is empty
        ((M)_{t+1})_0 = min_cost_aux(t + 1, T, s, C, S, c, 0, M)
    if h is 1
        return min(s + ((M)_{t+1})_1, S + c_t + ((M)_{t+1})_0)
    else if h is 0
        return min(C + s + ((M)_{t+1})_1, c_t + ((M)_{t+1})_0)
```

```
algorithm min_cost(I, s, C, S, tasks)
    M := new Matrix(|I| + 1, 2)
    c := new Array(|I|)
    c_i = 0 ∀i ∈ I
    j := 0
    foreach t ∈ I
        while j < |tasks| ∧ (tasks_j)_0 is t
            c_t = c_t + (tasks_j)_1
            j = j + 1
    return min_cost_aux(0, |I|, s, C, S, c, 0, M)
```

The foreach loop in the `min_cost` body costs T plus the number of tasks that are not alone in an instant, generally depends on the size of the tasks array.
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 it performs at most $2 * T + 2$ recursions and so the cost is linear, $O(T)$.
The final cost is so $O(|I| + |tasks|) = O(T + |tasks|)$.
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:

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

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:

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

So the minimum is always returned in every iteration.

Part 2

We can apply the considerations about the "not tasks" and the multiple tasks in a time instant of the previous point also to this problem.
With a cost matrix $CM_{T,K}$ we define the individual cost of a freelance of skill $k \in \{0...K\}$ at time $t \in \{0...T\}$ as follows:

$((CM)_t)_k = \Sigma_{j \in J_{t,k}} cost(j)$ with $J_{t,k} = \{j | time(j) = t \wedge k \in skills(j)\}$

With this consideration, we merged all the tasks at the same time in a single task with a different cost for each skill. This is possible due to the fact that if we have a hired worker with the skill k at time t he can process all the works at time t that require the skill k. So the choice is or use a hired worker for the skill k, or use all freelances with the skill k for the tasks at time t.
With a similar consideration as the one of problem 4.1, we insert costs 0 in $((CM)_t)_k$ when all tasks at time t do not require the skill k. If there are no tasks at time t we insert a row of all zeroes.
Now there is only a task for time instant and all tasks have the same number of individual costs, K.
We can solve the problem using a brute-force that does a recursion over all possible permutation of hired and freelance worker at each time instant. h is an array of K elements that associate the skill k to a boolean representing if we have a hired worker with the skill k or not.

$$helper(h, next, t) = \sum_{k \in \{0...K\}} \begin{cases} s & \text{if } h_k = true \wedge next_k = true \\ S + ((CM)_t)_k & \text{if } h_k = true \wedge next_k = false \\ C + s & \text{if } h_k = false \wedge next_k = true \\ ((CM)_t)_k & \text{if } h_k = false \wedge next_k = false \end{cases}$$

$$OPT(t, h) = min\Big\{OPT(t + 1, next) + helper(h, next, t) \ \forall \ next \in D'_{(2,K)}(true, false)\Big\}$$

The cost of this recurrence is $(2^K)^K * T$. We can memorize $OPT(t, next)$ in a matrix $M_{T,2^K}$ in order to gain a polynomial cost in function of 2^K. The first dimension represents the time instants, the second the dispositions with repetitions in $D'_{(2,T)}(true, false)$ (always with $true = 1$ and $false = 0$).
In pseudocode (tasks is an array of tuples (time instant, outsourcing cost, has skill 1 ... has skill K) and $I = \{0...T\}$):

```
1  algorithm skills_min_cost_aux(t, T, s, C, S, CM, h, M, K, disps)
2      if t is T
3          return 0
4      r := {}
5      for i = 0 to 2^K
6          if ((M)_{t+1})_i is empty
7              ((M)_{t+1})_i = skills_min_cost_aux(t + 1, T, s, C, S, CM, disps_i, M, K)
8          cost := ((M)_{t+1})_i
9          for k = 0 to K
10             if h_k = 1 ∧ (disps_i)_k = 1
11                 cost = cost + s
12             else if h_k = 1 ∧ (disps_i)_k = 0
13                 cost = cost + S + ((CM)_t)_k
14             else if h_k = 0 ∧ (disps_i)_k = 1
15                 cost = cost + C + s
16             else if h_k = 0 ∧ (disps_i)_k = 0
17                 cost = cost + ((CM)_t)_k
18         r = r ∪ {cost}
19     return min(r)
```

```
1  algorithm skills_min_cost(I, s, C, S, K, tasks)
2      M := new Matrix(|I| + 1, 2^K)
3      h := new Array(K)
4      CM := new Matrix(|I|, K)
5      disps := dispositions_with_repetitions((true, false), K)
6      for k = 0 to K
7          h_k = 0
8          j := 0
9          foreach t ∈ I
10             ((CM)_t)_k = 0
11             while j < |tasks| ∧ (tasks_j)_0 is t
12                 if (tasks_j)_{2+k} is true
13                     ((CM)_t)_k = ((CM)_t)_k + (tasks_j)_1
14                 j = j + 1
15     return skills_min_cost_aux(0, |I|, s, C, S, CM, h, M, K, disps)
```

The cost of the body of `skills_min_cost` is $O(K * (|I| + |tasks|))$ (note that it is more or less K time the loop in `min_cost`). The cost of the body of `skills_min_cost_aux` is $O(2^K * K)$ if we do not consider the recursive calls because we have a loop of K iterations in a loop of 2^K iteration. Not that the loop at line 9 can be avoided if we preprocess the value in a matrix. In order to get the number of recursion of `skills_min_cost_aux` we can count the number of entries that are filled in M because an entry is filled only one time when a new call is performed. The matrix size of $(|I| + 1) * (2^K)$ so the total cost of `skills_min_cost_aux` is $O(|I| * 2^K * 2^K * K) = O(|I| * 2^{2K} * K)$. The final cost is so $O(K * (|I| + |tasks|) + |I| * 2^{2K} * K) = O(2^{2K} * K * T + K * T + K * |tasks|)$. With

$L = 2^K$ it is $O(L^2 * log_2(L) * T + log_2(L) * (T + |tasks|))$ that is polynomial.
To prove that the algorithm is correct we proceed by induction.
In the base case $t = T - 1$, we consider all the dispositions with repetitions. These dispositions are all the possible configurations of hired or freelance for each worker types. So after computing the cost for each configuration, we can get the best configuration getting the minimal cost.
Now assume that at the iteration $t + 1$ returns the minimum cost. For the case t we are adding the cost of each configuration to the cost of the $t + 1$ case using such a configuration. This implies that we return the cost of the best configuration that creates a sequence of minimum cost.
So the minimum is always returned in every iteration.

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:

```
algorithm edge_is_in_mst_aux(G, e, v)
    r := true
    v.visited = true
    foreach i ∈ G.neighbors(v)
        if G.weight(v, i) >= G.weight(e)
            skip
        if i is e.u
            return false
        if not i.visited
            r = randedge_is_in_mst_aux(G, e, i)
    return r
```

```
algorithm edge_is_in_mst(G, e)
    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 [2]. The action of removing the edges with the minimum weight is done in constant time.
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 itarations of Kruskal using l as source of edges;

In pseudocode:

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

```
algorithm mst_from_edge(G, e)
    if not edge_is_in_mst(G, e)
        error()
    foreach v ∈ G.V
        make_set(v)
    ordered := sort_by_increasing_weigth(G.E)
    ordered = ordered \ e
    mst := {e}
    foreach o ∈ ordered
        if find_set(o.v) is not find_set(o.u)
            mst = mst ∪ {o}
            union(o.v, o.u)
    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 corretness 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 weigth $\leq$ of the weigth of T because the weigth of f cannot be $<$ of the weigth 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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define AT(mat, x, y) (mat)[(x)*2 + (y)]
#define MIN(a, b) ((a) <= (b) ? (a) : (b))

#define HIRED 1
#define FREELANCE 0

struct Task
{
    int time;
    int cost;
};

int min_cost_aux(int last_worker_type, int t, int T, int s, int C, int S, int* c, int* matrix)
{
  if(t == T)
    return 0;

  if(AT(matrix, t+1, HIRED) == -1)
    AT(matrix, t+1, HIRED) = min_cost_aux(HIRED, t+1, T, s, C, S, c, matrix);
  if(AT(matrix, t+1, FREELANCE) == -1)
    AT(matrix, t+1, FREELANCE) = min_cost_aux(FREELANCE, t+1, T, s, C, S, c, matrix);

  int x,y;
  if(last_worker_type == HIRED) {
    x = s + AT(matrix, t+1, HIRED);
    y = S + c[t] + AT(matrix, t+1, FREELANCE);
  }
  else {
    x = C + s + AT(matrix, t+1, HIRED);
    y = c[t] + AT(matrix, t+1, FREELANCE);
  }

  return MIN(x, y);
```

```
}


int min_cost(int T, int s, int C, int S, struct Task* tasks, int task_num)
{
    int* matrix = calloc(sizeof(int), (2*T +2));
    memset(matrix, -1, sizeof(int)*2*T);

    int* c = calloc(sizeof(struct Task), T);
    int i, j;
    for(i = 0, j = 0; i < T; ++i) {
        while(j < task_num && tasks[j].time == i) {
            c[i] += tasks[j].cost;
            ++j;
        }
    }

    int r = min_cost_aux(FREELANCE, 0, T, s, C, S, c, matrix);

    free(matrix);
    free(c);
    return r;
}

int main()
{
    struct Task tasks[] = {{0,3},{0,2},{1,6},{4,1}};
    int sol = min_cost(5, 2, 2, 3, tasks, 4);

    printf("%d\n", sol); //10
    return 0;
}
```

Exercise 5 Code

```
import networkx as nx

def is_in_mst(G, e):
    visited = [False]*(len(G.nodes()))
    e_w = G.edges[e]["weight"]

    def aux(v):
        visited[v] = True
        r = True

        for i in G.neighbors(v):
            if G.edges()[v,i]["weight"] >= e_w:
                continue
            if i == e[1]:
                return False
            if visited[i] == False:
                r = r and aux(i)

        return r

    return aux(e[0])


def kruskal(G, e=None):
    # e!=None forces to include edge e in the MST
    # otherwise standard kruskal is performed

    def find(parent, i):
        if parent[i] == i:
            return i
        return find(parent, parent[i])

    def union(parent, order, x, y):
        rx = find(parent, x)
        ry = find(parent, y)
        if order[rx] < order[ry]:
            parent[rx] = ry
        elif order[rx] > order[ry]:
            parent[ry] = rx
        else :
            parent[ry] = rx
            order[rx] += 1
```

```python

    mst = []

    i = 0
    j = 0

    s_edges = sorted(G.edges(), key=lambda x: G.edges[x]['weight'])
    if e is not None:
        s_edges.remove(e)
        s_edges = [e] + s_edges

    parent = []
    rank = []

    for n in G.nodes():
        parent.append(n)
        rank.append(0)

    while j < len(G.nodes())-1:
        if i == len(s_edges): break

        u,v = s_edges[i]
        w = G.edges[u,v]['weight']

        i += 1
        x = find(parent, u)
        y = find(parent, v)

        if x != y:
            j += 1
            mst.append((u,v,w))
            union(parent, rank, x, y)

    return mst

def mst_from_edge(G, e):
    if not is_in_mst(G, e):
        raise Exception("edge %s cannot be in a MST" % str(e))
    return kruskal(G, e)
```

4.2 Solution running K times 4.1

```
1  algorithm skills_min_cost2(I, s, C, S, K, tasks)
2      r := 0
3      for k = 0 to K
4          c := new Array(|I|)
5          j := 0
6          foreach t ∈ I
7              c_t = 0
8              while j < |tasks| ∧ (tasks_j)_0 is t
9                  if (tasks_j)_{2+k} is true
10                     c_t = c_t + (tasks_j)_1
11                 j = j + 1
12         M := new Matrix(|I| + 1, 2)
13         r = r + min_cost_aux(0, |I|, s, C, S, c, 0, M)
14     return r
```

The cost of the body of the loop at line 3 is $O(|I| + |tasks|)$ without considering the call to `min_cost_aux`. The cost of `min_cost_aux` is $O(|I|)$ as shown in 4.1. This implies that the total cost of `skills_min_cost2` is K times the cost of the loop at line 3, $O(K * (2 * |I| + |tasks|))$

Can this formulation of the algorithm be correct? I would like to discuss a few minutes about this question during the oral exam.

References

[1] "Approximation algorithm - Wikipedia." `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)`. Accessed: 2018-11-20.

[3] "Minimum spanning tree, Cycle Property - Wikipedia." `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`. Accessed: 2018-11-14.

[5] "Timsort - Wikipedia." `https://en.wikipedia.org/wiki/Timsort`. Accessed: 2018-11-14.