Homework 1

Algorithm Design 2018-19 - Sapienza

Luigi Russo 1699981

December 7, 2018

Contents

Introduction

For every problem I provide the solution algorithm, the running time and a proof of correctness. At the end of the document there is the list of external resources that I have used. I have also implemented in Python 3.7 the following:

- Exercise 1 (*)
- Exercise 4.1
- Exercise 4.2 (*)
- Exercise 5.1
- Exercise 5.2

(*) actually not required

1 Cristina and centers

Cristina is interested in a metric space (X, d), where all distances d(x, y) are either 0 (in which case x = y), 1 or 3. Further, all distances are symmetric and obey the triangle inequality, that is $d(a, b) + d(b, c) \geq d(a, c)$. Cristina wants to cluster the points of X. Unfortunately, she does not yet know how many clusters she has to use. She therefore will find a permutation $\Pi(X)$ and use, for every $k \in \{1, ..., |X|\}$, the first k elements $C = \pi(X)_1, ..., \pi(X)_k$ of the permutation as centers.

Goal: For any such k, the output should be optimal with respect to following objective function:

$$\max_{x \in X} \min_{c \in C} d(x, c). \tag{1}$$

Hint: Use a greedy algorithm. The running time should be no larger than $O(|X|^2)$

1.1 The algorithm

This problem can be seen as a variant of the well-known [1] K-centers decision problem. I have designed a greedy algorithm that solves the above problem. It creates (and returns to Cristina) the desired permutation in this way.

1. $C := [\,]$
2. pick a random point c_first and add it to the list of centers C.
3. for every point v ∈ V (v $\notin C$), compute the minimum distance from C to v
4. pick the point c_next with highest distance from the points in C
5. add c_next to C, "relax" the distances of points outside C, and continue the process from (3) until $|V| = |C|$

1.1.1 Source-code

I have implemented the algorithm in Python 3.7 and the source code is attached as ex1.py. Moreover, I provide the pseudo-code as ex1.txt.

1.1.2 Running time

The algorithm adds $|X|$ points to C. How much costs adding a point to C? A point is added to C if, among all the points v $\notin$ C, it has the maximum distance from the ones in C. At each iteration we update the minimum distance from all points to C, simply considering if the new center can "decrease" the current distance of the point from C. It costs $|X-1|$ for the second center, $|X-2|$ for the third and so on. Since the iterations are $|X|$ and each one costs $O(|X|)$ we have the cost is $O(|X^2|)$.

1.1.3 Proof of correctness

At each iteration the algorithm updates (if possible) the current maximum distance from the list of centers. This leads to a non increasing sequence of max_values $= \{d_1, d_2, ..., d_{|X|}\}$. Given an optimal solution O, we have to prove that the sequence computed with the greedy algorithm above is such that for every $i \in \{1, 2, ..., |X|\}$, $d_g[i] = d_o[i]$, where $d_g[i]$ is the maximum distance from C in the greedy algorithm and $d_o[i]$ is the maximum distance from C in the optimal solution at iteration i. Since my greedy algorithm guarantees [1] that for every k, $d_g[k] \leq 2 \cdot d_o[k]$ and since the only possible values of d are 0, 1 or 3, we have these possible cases:

- $d_o[k] = 0 \Rightarrow 0 \leq d_g[k] \leq 2 \cdot 0 = 0 \Rightarrow d_g[k] = 0$ OK
- $d_o[k] = 1 \Rightarrow 1 \leq d_g[k] \leq 2 \cdot 1 = 2 \Rightarrow d_g[k] = 1$, OK since 1 is the only value that d can assume s.t. $1 \leq d \leq 2$
- $d_o[k] = 3 \Rightarrow 3 \leq d_g[k] \leq 2 \cdot 3 = 6 \Rightarrow d_g[k] = 3$, OK since 3 is the only value that d can assume s.t. $3 \leq d \leq 6$

And this is true for every k.

2 Streets and avenues

Consider a city with m parallel horizontal streets and n parallel vertical avenues. These lines cross in $m \times n$ intersections. On $k \in \{1, ..., m \times n\}$ of these intersections, special checkpoints are placed. We want to place video cameras on a subset of the streets and of the avenues such that each checkpoint is in the visibility range of a camera. A camera allows to monitor all the checkpoints of an avenue or of a street. The subset of may contain both horizontal streets and vertical avenues. Clearly, you can always select all m + n of these streets and avenues. The challenge therefore is to select a smallest subset of these streets and avenues such that each checkpoint is in the visibility range of a camera.

2.1 The algorithm

1. Consider the starting grid as the biadjacency matrix of a graph $G = (S, A)$, where S is the set of the streets and A is the set of the avenues: each checkpoint marks the presence of an edge of capacity 1.

2. Create a super-source node connected to all the nodes in S and a super-sink node connected to all the nodes in A. The capacity of each of these edges is set to 1. Call G' such a graph.

3. Run Ford-Fulkerson algorithm to find the minimum-capacity cut C.

4. Define $L1 := L \cap C$, $L2 := L - C$, $R1 := R \cap C$, $R2 := R - C$

5. Let B be the set of nodes in R2, s.t. there is some edge from them to L1.

6. Return as output the set $O := L2 \cup R1 \cup B$

2.1.1 Running time

Finding the minimum capacity cut in the first part costs only O(nm) if we use Ford Fulkerson. Once we compute the cut with minimum capacity we just have to make some intersections between L, R and C. In particular: L1 and L2 can be computed in O(m), R1 and R2 in O(n) and also B can be computed considering the edges between L1 and R2; remember that the

maximum number of edges can be nm. So the total cost of the algorithm is bounded to O(nm).

2.1.2 Proof of correctness

The starting schema can be seen as the biadjacency matrix of a graph G = (S, A), where S is the set of the streets and A is the set of the avenues. Every checkpoint can be interpreted as an edge (u,v), with $u \in S$ and $v \in A$. What we want to find is the so called vertex cover, i.e. a set of nodes such that each edge of the graph has at least one endpoint in the set. The output set O covers all edges that have an endpoint either in L2 or R1, because O includes all nodes of L2 and all nodes or R1. The nodes that have endpoint in L1 and the other endpoint in R2 are covered by B. This means that O covers all the nodes "involved" in the checkpoints. Moreover, there is no other set, with lower cardinality, that satisfies this property. In fact, let k be the capacity of the minimum capacity cut. Then $k = |L2| + |R1| + |edges(L1, R2)|$ and so $k > |L2| + |R1| + |B| = |O|$. But k is equal to the capacity of the minimum cut in G', which is equal to the maximum flow in G'. This quantity is also equal to the size of the maximum matching in G [2]. This means that G has a matching of size k, and so every vertex cover must have size $s \geq k \geq |O|$. And this proves that k is exactly the size we were looking for.

3 Birtday NP

Michele's birthday is coming up and he is thinking about who to invite for the party. He has male friends, denoted by the set M, and female friends, denoted by the set F. Between any two friends $x, y \in M \cup F$, there exists a score $w(x, y) \in \{0, 1\}$. The score tells us whether the two friends like each other or not (0 meaning that they do not like each other, 1 meaning that they like each other). For the party to be successful we have to consider the following two constraints.

- Michele would like to maximize the number of liked guests over the total number of invited guests. Formally, let $I \subseteq M \cup F$ be the invited guests. Then we want to maximize $\frac{1}{|I|} \sum_{x,y \in I} w(x, y)$
- Michele insists upon having an equal number of female and male guests. That is $|I \cap M| = |I \cap F|$.

Goal: Show that this problem is NP-complete.

3.1 Proof

3.1.1 Formalization as graph problem

We can represent each friend as a node in an undirected graph G. We also add an edge between two nodes x and y if and only if w(x,y) is equal to 1. In this case the function we want to maximize is exactly the density [3] of a subgraph of G.

3.1.2 Overview

First of all let's notice that without the second constraint (number of males equal to number of females) the problem can be solved in polynomial time [3], since the first requirement is asking for the densest subgraph of G. I am going to show that adding the second constraint, the problem can be reduced to K-clique problem, that is a well-known NP-complete problem. I am going to define a variant of the above problem, that can simplify my proof. Let's call A the original problem, B the K-clique problem, and C the following: given a number K, find a subgraph with density (at least) K and an equal number of M and F nodes. Problem A can be trivially reduced to problem C, because fixing a number $\bar{\text{K}}$ we can check if problem C has a solution. If

yes, we can increment $\bar{K}$, otherwise we can decrement it. After a certain number of iterations (see next for how many exactly) the problem C finds a solution for the maximum K: this solution is what the problem A is asking for. How many steps are needed to find the max K? We can start fixing $\bar{K} = |V|$. If a solution is found, of course it is optimal for A; otherwise, we behalf $\bar{K}$ and retry, thus adopting a sort of *binary search of K*, in the sense that we proceed in this way to find in a logarithmic (in function of $|V|$) number of steps the right K. So, finding a solution for problem A with this method simply increases the total cost of a factor $O(log|V|)$. It's time to proof that C can be reduced to B. If we are able to do this, we can conclude that also A can be reduced to B.

3.1.3 NP certifier

We can easily verify if an instance S is a solution of problem C. Indeed, we can verify if the number of males is equal to the number of females, and this can be done simply looking at the nodes of S. If we also count the number of edges in S, we can compute the density of S and check if it is equal to K. Now we can focus on the reduction.

3.1.4 Reduction: Step 1

Problem B asks for a clique of size K_B. Let's label all the nodes of G as males (the M set). At this point we can add a clique (we forge it, adding new nodes and all the necessary edges to the graph) of size K_B, all made up by females (i.e. the F set). This can be done in polynomial time, since we just have to add $K_B \leq |V|$ nodes and a number of edges that is a polynomial function of $|V|$ (quadratic, to be precise).

3.1.5 Reduction: Step 2

Let's fix $K_C = \frac{K_B - 1}{2}$. Let's call ALG the algorithm that solves the problem C, and be SOL its solution. At this point we can run ALG in order to find a solution SOL with density K_C. The solution, if any, **must** contain the whole clique of size K_B of females (we added previously) and a clique of size K_B of males. Remembering that the number of edges in a clique of n nodes is $\frac{n(n-1)}{2}$, the density of such a subgraph is, how we can expect, equal to $\frac{\frac{K_B(K_B-1)}{2} + \frac{K_B(K_B-1)}{2}}{2 \cdot K_B} = \frac{K_B(K_B-1)}{2 \cdot K_B} = \frac{K_B - 1}{2}$, that is exactly the number we fixed

as K_C.
Is it possible that the solution found is not the union of two such cliques? Let's consider some cases. Of course the number of male and female nodes must be the same, otherwise this is not a solution for problem C. Moreover, we know that the maximum number of female nodes is K_B, since we forged them at step 1. This means the that also the maximum number of male nodes in SOL must be equal to K_B. Imagine that SOL is a clique with density K_C and a number of male nodes strictly lower than K_B. This means that also the number of female nodes in SOL is strictly lower than K_B. But this leads to a density strictly lower than K_C, thus resulting in an absurd. And we also find an absurd, if we say that SOL is not the union of two cliques, and in particular, the male nodes do not form a clique, because the number of edges in this case would be strictly lower than what we are expecting, with a density strictly lower than $\frac{K_B-1}{2}$.

3.1.6 Reduction: Step 3

Problem B now can be solved, simply deleting from SOL all the females (forged) nodes, thus obtaining a clique of size K_B, all made up by male nodes of course. This step, as step 1, requires polynomial time. In particular, the cost is bounded to $O(|K_B|) \leq O(|V|)$ to delete nodes and becomes quadratic if we also remove the edges.

4 Hiring process

1. A consulting company can execute tasks requested from its customers either with hired personnel or with freelance workers. The set of tasks is presented on a subset S of time instants {1, ... T}. If task j_t is assigned to a hired employee of the consulting company, the cost is given by his daily salary s. If task j_t is outsourced to a freelance worker, the paid cost is c_t and it depends on the specific task and time instant. A worker can be hired at any time by paying him a hiring cost C and fired at any later time by paying a severance cost S.
   Goal: Design an optimal strategy that runs in polynomial time that minimizes the total cost of executing the tasks. The total cost should include the costs paid to the freelances workers and the costs paid for hiring, firing and the salaries of the hired personnel. Prove that the algorithm is correct and provide an analysis of its running time. Implement the algorithm with a programming language of your choice.

2. Assume now that task j_t requires a set $W_t \subseteq W, |W| = k$, of a constant number k of different types of workers. The company should therefore decide which types of workers to hire and which types of workers to outsource. The salary cost for any time instant is the same for all types of workers as well as the hiring and the firing cost. The cost of worker $j \in W$ varies with time: the cost of worker $j \in W$ is equal to c_t^j.
   Goal: Design an optimal strategy that runs in polynomial time that minimizes the total cost of executing the tasks.
   Hint: Use dynamic programming for both exercises. The polynomial running time of the final algorithm in the second exercise should depend on T and on 2^k. Start with the case k = 2 an generalize the approach from there.

4.1 The algorithm

Given a sequence of tasks, ordered by period t, the algorithm creates a matrix of costs. There are 2 rows and T columns: matrix[r][c] is the minimum cost if you decide to have r workers in period c, where r is either 0 or 1. This matrix is built for each period t, considering which is the best option of the previous period that leads, with all the necessary costs, to the current one. For example, matrix[1][5] can derive from two previous cases: matrix[1][4]

and matrix[0][4]; in the first case we have just to pay the daily salary, since the worker has already been hired, instead in the second one we should also pay the hiring cost.

1. For every period $t \in \{1, ..., T\}$ take all tasks that have to be processed in period t.
2. There are two possible cases:
   (a) assign all the tasks to a worker. For sure we have to pay the salary for period t; add to it the minimum between
      - matrix[0][t-1] plus the hiring cost
      - matrix[1][t-1]

   (b) assign all the tasks (if any) to freelancers. Pay all the outsource costs; add to this the minimum between
      - matrix[0][t-1]
      - matrix[1][t-1] plus the severance cost
3. return the minimum between matrix[0][T] and matrix[1][T]

4.1.1 Source code

I have implemented the exercise in Python language. The code is attached as ex4_1.py.

4.1.2 Running time

There are T iterations: at each of these the algorithm collects all the tasks belonging to that period and makes a binary decision, based on the costs already computed at the previous iteration. The cost per period, indeed, is O(1) since it is based on precomputed values. The overall cost of the algorithm is O(T).

4.1.3 Proof of correctness

For each period t there are two possible cases: either we have a hired worker who can process all the tasks, or we do not have him: in this case, if there is some task to be processed, it has to be assigned necessarily to freelancers. The algorithm minimizes the cost of having 0 or 1 worker for every period,

basing its analysis on the previous computations. The base case is for t = 0, where the minimum cost is equal to 0 (since the first task is at least at period t=1). The algorithm finds the same solution, since it picks the minimum between 0 (no hire) and some cost $c > 0$ (hire someone and pay him). Moreover, c is the minimum cost of having paid personnel in period 0. So for period t = 0 the algorithm correctly computes the two minimums and in particular the optimal solution. Now, suppose the algorithm correctly computes the minimum costs $opt[*][i]$ for a given period i and both the possible cases (hired worker for period i, or not). At period t = i+1, the algorithm computes the minimum between having or not a paid worker, considering **all the possible "paths"** from all the previous optimal solutions (possible hirings, firing, salaries, etc.), thus resulting in optimal solutions at period i+1. This can be easily deduced by applying the following recursive formulation:

- m[0][t] =
    1. t = 0: 0
    2. $t > 0$: outsource(all tasks of period t) + min(m[0][t-1], m[1][t-1] + severance)
- m[1][t] =
    1. t = 0: hire
    2. $t > 0$: salary + min (hire + m[0][t-1], m[1][t-1])

with $\forall t, OPT(t) = min(m[0][t], m[1][t])$

m[0][i] and m[1][i] are optimal[1], and are the two only possible scenarios for period i. Also for period i+1 I can have only two scenarios, each of these can "derive" from one of the two previous. Since we only take the necessary costs, and we take the minimum of the two, m[0][i+1] and m[1][i+1] are still optimal. *We could also proof this by contradiction, assuming that until iteration i the solutions are optimal and at iteration i+1 the algorithm makes a "wrong" decision. But we know it has considered all the paths from previous optimal solutions, adding only the "necessary" cost, and nothing more. This means that some previous solution at period i was not optimal, thus resulting in an absurd.*

[1] in the sense that they are the two minimum at period i. In general they are different quantities, so that the minimum of them is the actual optimal cost.

4.2 The algorithm

I have designed and implemented 3 algorithms[2] to solve the problem above: the first one (ex_4_2_normal.py) is the generalization of the algorithm 4.1 and considers multiple freelancers for each task, meaning that also freelancers have exactly one skill as the other workers. This was my original algorithm (what I have understood at beginning). This algorithm can be also reformulated in a more efficient way, iterating k times the 4.1 algorithm: this is the second algorithm (ex_4_2_optimized.py). Finally, I have written a third algorithm (ex_4_2_variant.py), considering that the outsource cost is just a fixed amount, meaning that we do not outsource the skill but the task itself! (so it's like the freelancer is a super worker with all the necessary skills): this last, as will be pointed out in the next sections, can be also solved in $O(T \cdot 2^k)$, but not in $O(k \cdot T)$.

4.2.1 Generalization of problem 4.1

Given a sequence of tasks, ordered by period t, and a number of skills k, the algorithm creates a matrix of costs. There are 2^k rows and T columns: matrix[mask][c] is the **minimum cost** if you decide to have $\#workers(mask)$ in period c, where mask represents a combination of hired workers (e.g. "0111" means that we have hired workers with skills 1,2,3 and no hired worker with skill 0). This matrix is built for each period t, considering which is the best option of the previous period that leads, with all the necessary costs, to the current one. Again, the minimum cost has to take in consideration possible firings, hirings and outsource costs.

1. For every period $t \in \{1, ..., T\}$ take all tasks that have to be processed in period t.
2. There are 2^k possible cases, corresponding to different masks (combinations of hired workers): in particular, we could have or not a worker with a skill $s_i, i \in \{1, ..., k\}$.
   (a) for each of these combinations, minimize the cost for a given mask and a given period, based on the previous costs.

[2]Due to the "mess" on Piazza Q & A, I have decided to list here more than a solution, each one with some different assumption

3. return the minimum between matrix[$mask_1$][T], matrix[$mask_2$][T], ... matrix[$mask_{2^k}$][T]

4.2.2 Optimized version

This algorithm just consists of running the 4.1 k times, and summing up all the minimum costs found at the end.

4.2.3 The variant with "super" freelancer

The algorithm is actually quite similar to the first one: there are 2^k choices we can do at each iteration (period). The only difference is in the function that computes the cost of freelancers, since the outsource cost is just a fixed quantity and not a value specific per task. Moreover, outsourcing a task may lead to hire no other worker, since all the necessary skills are "covered" by the freelancer.

4.2.4 Source code

I have implemented the exercise in Python language. The code of the 3 algorithms is attached.

4.2.5 Running time

The number of iterations is always T (as in 4.1). What changes is the number of possible choices we can make at each step: this is 2^k for both first and third algorithm. Since these choices all use precomputed values, this means that the cost per choice is $O(2^k)$, for an overall $O(T \cdot 2^{2k})$. In the optimized version we just run k times the algorithm 4.1, thus reducing the cost to $O(k \cdot T)$.

4.2.6 Proof of correctness

This can be seen as a generalization of the previous problem. Again the algorithm explores **all the possible paths** that minimize for every period and every scenario (what I have called mask), computing the optimal solution. This can be proved by induction, since the base case is easily exploitable (the algorithm computes 2^k costs, considering only the necessary cost) as done for algorithm 4.1: the fact that there are more choices does not alter the correctness, since all the subproblems are taken into account.

$\forall t \in \{0, ..., T\}, OPT(t) = min_{i \in \{1,...,k\}} m[i][t]$

It could be proved by contradiction: assuming that until iteration i the solution is optimal (and also all the minimum costs are optimal for that given scenario and period), and that all the possible paths from period t and period t+1 are taken into account, a non optimal solution at iteration i+1 would lead to an absurd (as in the case with k=1). As for the optimized version, it is based on the 4.1 correctness. Since the costs per skills are *independent* each others (there is no super freelancer!) we can simply minimize the costs for each skill. Say the solution is not optimal: this means that, for some skills, we made a wrong decision. But this cannot be true, since we have actually minimized the cost considering one skill at time.

5 Federico and MST

Federico is a mathematician who doesn't like stories and wants the exercise to get to the point.
Goal: We are given a weighted graph G(V, E). Let e $\in$ E.

1. Design an algorithm that decides whether or not there exists a minimum spanning tree containing e. For full marks, the algorithm must run in time at most $O(|V| + |E|)$.

2. Design an algorithm that computes a minimum spanning tree containing e, if one exists. For full marks, the algorithm must run in time at most $O(|E|log|E|)$. Implement the algorithm with a programming language of your choice.

5.1 The algorithm

Given $e = (u,v)$:

1. run a DFS from the endpoint u **considering only** those edges (a,b) s.t. weight$(a,b) <$ weight(e).

2. Two possible cases:

   (a) If during the DFS there is an edge that leads to node v, then the edge e does not belong to any MST.

   (b) If the DFS terminates and case 1 never happens, then there exist some MST that contains e.

5.1.1 Running time

This is a simple variant of a DFS on a graph: the fact that we "filter" the edges can lead even to a reduction of visited nodes. In the worst case we know that DFS costs $O(|V| + |E|)$ and this is still the upper bound of algorithm 5.1.

5.1.2 Source code

See next sections.

5.1.3 Proof of correctness

I have used the MST cycle property in this way: we know that given a cycle in a graph, the largest edge among the ones that form the cycle cannot belong to a MST. So, the DFS run above tries to connect u and v with edges **strictly lower** than e: if this happens, it means that there exist some cycle in the graph that connects u and v and, of course, such cycle contains the edge e: since all the edges considered during our *custom* DFS are strictly lower than e, we can conclude that e is the largest edge in a cycle of the graph. So, by the MST cycle property, e cannot belong to any MST. Since this condition is necessary and sufficient, this is enough for our proof.

5.2 The algorithm

1. Run the algorithm 5.1: if e cannot belong to any MST throw an error.
2. Run a *custom* Boruvka, whit all disconnected components, except u and v: we connect them since the beginning adding edge e to the MST.

5.2.1 Source code

I have implemented the exercise in Python language, using networkx library for graph management (add nodes, get adjacent nodes, etc.). The code is attached as ex5.py.

5.2.2 Running time

The first part of the algorithm is run the 5.1, whose cost is $O(|V| + |E|)$. In the second part we perform a custom Boruvka, that is actually a Boruvka after a single union operation. This means that we can fix the bound of the algorithm to the well-known one of Boruvka: $O(|E|log|V|)$. If we assume that the graph is connected this is $\leq O(|E|log|E|)$.

5.2.3 Proof of correctness

The algorithm 5.1 is able to verify that a MST with edge e really exists. Once we know that such MST can be built, we run a custom version of Boruvka algorithm. Since we know that a MST with edge e exists, we also know that

u and v belong to the same connected component; moreover, we know that the edge e must be part of the MST and e is the edge (the only one of course in the MST we compute) that has to connect u and v. So we start Boruvka initializing all the nodes (except u and v) as disconnected components: as for u and v, we bind them to the same connected component (with edge e). Now, Boruvka algorithm will run and produce a MST: we can be sure about that because this algorithm is designed to find a MST starting from a forest of disconnected components. **Note:** The algorithm will fail if the graph is not connected actually, but we can assume the graph is always connected, or we can check easily at the end of the algorithm if the current MST is really a tree: the number of edges *returned* must be n-1.

References

[1] *Wikipedia: Metric K-center*
https://en.wikipedia.org/wiki/Metric_k-center

[2] *Wikipedia: Konig's theorem*
https://en.wikipedia.org/wiki/K%C5%91nig%27s_theorem_(graph_theory)

[3] *Wikipedia: Dense Subgraph*
https://en.wikipedia.org/wiki/Dense_subgraph