Homework 1

Algorithm Design 2018-19 - Sapienza

Luigi Russo 1699981

December 1, 2018

Contents

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 $d(a, b) + d(b, c) >= d(a, c)$. 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 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 c1 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 c2 with highest distance from the points in C
5. add c2 to C and continue the process until $|V| = |C|$

1.1.1 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 $O(|X^2|)$

1.1.2 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. Since my greedy algorithm guarantees that for every k, $d_g[k] \leq 2 * d_o[k]$ 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 cost of the maximum flow in G' which is equal to the size of the maximum matching in G (for Konig theorem). This means that G has a matching of size k, and so every vertex cover must have size $>= k >= |O|$.

3 Birtday NP

Reduce to Crique problem.

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

4.1.1 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).

4.1.2 Source code

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

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.

4.2 The algorithm

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 s1.
   (a) for each of these combinations, minimize the cost for a given mask and a given period, based on the previous costs.
3. return the minimum between matrix[$mask_1$][T], matrix[$mask_2$][T], ... matrix[$mask_{2^k}$][T]

4.2.1 Running time

The number of iterations is always T (as in the first algorithm). What changes is the number of possible choices we can make at each step: this is 2^k. Since these choices all use precomputed values, this means that the cost per choice is constant O(1), for an overall $O(T \cdot 2^k)$.

4.2.2 Source code

I have implemented the exercise in Python language. The code is attached as ex4_2.py.

4.2.3 Proof of correctness

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 that have weight less than that of 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 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 $\leq O(|E|log|E|)$.

5.2.2 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.

5.2.3 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.