Algorithm Design - Homework 1

Francesco Carpineti 1683418
Federico Gioia 1702089
Giuliano Abruzzo 1712313

7-12-2018

Contents

1 Exercise 1

1.1 Problem

We are given a metric space *(X, d)* where all distances *d(x,y)* are either 0, 1 or 3. These distances are symmetric and obey the triangle inequality. The goal is to cluster the points of X without knowing how many clusters to use. We do this finding a permutation of points $\pi(X)$ such that, choosing an integer k from 1 to $|X|$, I can use the first k elements of $\pi(X)$ as centers. These centers are calculated in such a way as to minimize the maximum distance between each point and its closest center, so to minimize the covering radius for that value of k.

1.2 Algorithm

In order to find the permutation we just mentioned, I used this algorithm:

```
find_permutation(X):
  P = empty array of |X| elements;
  X1 = empty array of |X| elements;
  X3 = a copy of X array, given as input;
  for k=1 to |X|:
      c = pick_center(X1,X3);
      P[k]=c;
      for x in X3:
          if distance(x,c) == 1:
              add x to X1;
              remove x from X3;
  return P

pick_center(X1,X3):
  if X3 is not empty:
      x = first element of X3;
      remove x from X3;
  else:
      x = first element of X1;
      remove x from X1;
  return x;
```

This algorithm simply chooses the point farthest away from the current set of centers in each iteration as the new center. In an execution, the algorithm will put points in permutation P in this order: first, it takes a point from X3 (initially a copy of X) putting it into P and a nested for cycle puts all the other points of X3 with a distance of 1 from the former point into X1. This is repeated until X3 is empty. Second, all the points contained in X1 are added to the permutation (the algorithm won't execute the nested for cycle anymore). The

returned permutation P satisfies our constraint for every k-sized subset taken from its beginning, where k is an integer from 1 to |X|.

1.3 Proof of correctness

This algorithm uses a greedy approach, so we are trying to find an approximation to the optimal solution in small steps. So, the covering radius *r(C)* of our case will satisfy this relation: $r^{opt}(C^*) \leq r(C) \leq 2*r^{opt}(C^*)$. This can be justifiable thinking about a set X with two points only. The optimal algorithm would set a center in the perfect half of the two points, while a greedy algorithm would set a center on one of the two points. This means that in the greedy case, a point will be at distance $2*r^{opt}(C)$ from the other (set as center), while in the optimal case every point is at $r^{opt}(C)$ distance from the center. Let's prove this by contradiction. Suppose that the distance from the farthest point to all centers is $> 2*r^{opt}(C)$. This means that all distances between centers are also $> 2*r^{opt}(C)$. We have k+1 points (k centers found by our algorithm plus the point we just mentioned) with distances $> 2*r^{opt}(C)$ between every pair. Each point has a center of the optimal solution with distance $\leq r^{opt}(C)$ to it obviously. There exists a pair of points with the same center in the optimal solution for the pigeonhole principle, so the distance between them should be at most $2*r^{opt}(C)$ for the triangle inequality, which is a contradiction with what we said before for the k+1 points. In our study case, distances are defined so we can simply replace our values in the inequality explained at the beginning of this paragraph. There are four cases in our algorithm:

- *r(C)* == 0, which coincides with the optimal value;
- *r(C)* == 1 and $r^{opt}(C^*) == 3$, which is not possible cause our algorithm cannot return a value better than the optimal one;
- *r(C)* == 3 and $r^{opt}(C^*) == 1$, which is not possible cause the inequality would not be respected (in particular we would have 3 ≤ 2*1, obviously not possible);
- *r(C)* == $r^{opt}(C^*)$, so 1==1 or 3==3, which is great cause i reached the optimal value.

These cases show that our algorithm finds the optimal value and not an approximation.

2 Exercise 2

2.1 Problem

We have m streets and n avenues. We want to place videocameras on a subset of the streets and avenues in a way such that checkpoints present in some intersections between streets and avenues are recorded by them. A videocamera allows to monitor all checkpoints of an avenue or a street. Clearly, it is possible to put cameras in every street and avenue, but we want to put the smallest number of videocameras such that each checkpoint is under the visibility range of one of a camera.

2.2 Algorithm and proof of correctness

Obviously, we can consider streets and avenues as a matrix, where streets are rows and avenues are columns. An entry of this matrix will be checked if there is a checkpoint in the intersection between that street and avenue, unchecked otherwise. We can see our matrix as a graph, composed as follows:

	a1	**a2**	**a3**	**a4**	**a5**	**a6**
s1			X			
s2		X	X		X	
s3			X	X		
s4			X			

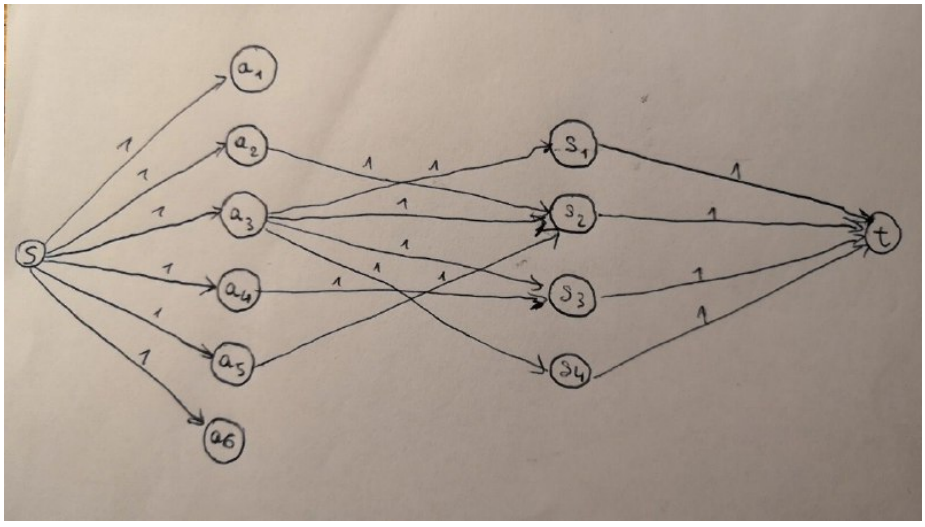

Figure 1: Graph for the matrix.

As we can see, if an avenue is connected to a street it means that there is a checkpoint on the intersection between them. We added source and sink in order to use Ford Fulkerson algorithm in a correct way. All the edges have a capacity of 1. We have to apply this algorithm on the graph, so we obtain the maximum flow of it. After, we have to remove source and sink (with incident edges too), so the remaining part of the graph is a bipartite graph, composed by streets and avenues only. Let's say L are the avenues on the left, and R the streets on the right (we will use this notation). The maximum flow returned by FF algorithm corresponds to the maximum matching M, subset of edge set E, on the bipartite graph, hence we know that in our maximum matching every street and avenue (nodes of our bipartite graph) is present in at most one edge. There is a theorem which says this: 'Maximum matching M cardinality in a graph is equal to maximum flow of that graph'[1]. So, edges selected by FF algorithm are the same of M. Now, we have to decide in which street or avenue at the ends of M to put videocameras. We can do this using Konig Theorem's proof[2]. Konig Theorem says that the cardinality of M is the same as the vertex cover one. So, using its proof, we can define how the vertex cover is composed and hence where we have to put videocameras. No vertex in a vertex cover can cover more than one edge of M, so if a vertex cover with |M| vertices can be constructed, it must be a minimum cover. To construct such a cover, let U be the set of unmatched vertices in L (it can even be empty) and let Z be the set of vertices that are either in U or are connected to U by alternating paths (paths that alternate between edges in M and edges not in M). Let $K = (L \setminus Z) \cup (R \cap Z)$. Every edge e in E either belongs to an alternating path or it has a left endpoint in K. If e is matched but not in an alternating path, then its left endpoint cannot be in an alternating path and thus belongs to $L \setminus Z$. Alternatively, if e is unmatched but not in an alternating path, then its left endpoint cannot be in an alternating path, for such a path could be extended by adding e to it. Hence, K forms a vertex cover and it contains vertices where we have to put videocameras. The running time analysis for the algorithm is restricted to the FF algorithm for the maximum flow and Konig proof for the identification of minimum vertex cover. Since (as we described previously) we adopt a union-find data structure, find and union operations done in Konig theorem's proof have unitary time of execution. Thus, the running time is strictly related to FF implementation that is O(m*f), where m is the number of edges and f the maximum flow starting from the source s.

[1] http://www.disrv.unisa.it/professori/anselmo/25_algo_Flusso_II_lezione.pdf
[2] https://en.wikipedia.org/wiki/K%C5%91nig%27s_theorem_(graph_theory)

3 Exercise 3

3.1 Problem

We are given two sets, one for males M={*m1, m2, ...*} and one for females F={*f1, f2, ...*} where each pair in M∪F has a score which can be either 0 or 1, depending on whether the two people like each other or not. We can see this as a weighted graph G with two different types of vertices (M for male, F for female). Weights on edges can be either 0 or 1 depending on the score (the graph is undirected and weights are symmetric): for an easier representation of the problem, edges mean that two people(vertices) like each other, no-edges otherwise. The problem consists of finding a subgraph of G where two constraints are satisfied:

- Weight sum is maximized over the total amount of nodes;
- The number of male vertices equals the female one.

The goal is to show that this problem is NP-complete. To do it, we have to verify these two properties:

- Given any candidate solution SOL, I can check if SOL satisfies problem constraints in poly-time;
- Our problem is at least as hard as another NP-complete problem.

3.2 Proof

To verify the first property, it is sufficient to use an algorithm that, given any solution SOL and an integer k, does a DFS on SOL (recall that SOL is a graph) keeping count of four integers: number of males, females, vertices and edges. At the end of it, considering that each vertex and each edge is visited only once, we can check if males and females are present in the same quantity and verify that graph density is at least k (graph density is defined as |E|/|V|). This can be done obviously in polynomial time, cause we use a simple DFS on a graph. For the second property, we have to choose an NP-complete problem, say X, and prove that X *polynomial reduces to* our problem, say Y. This means that arbitrary istances of X can be solved using a polynomial number of standard computational steps and a polynomial number of calls to a *black-box*, i.e. an algorithm that we suppose can solve our problem Y in poly-time. In our study case, we consider k-clique decision problem as X and we polynomial reduce it to our problem. How can we do this? First, we take an instance *I(X)* of X and transform it in an instance of Y, say *I(Y)*. We choose, as *I(X)*, a not-weighted, not-oriented graph composed by n vertices with inside a k+1 vertices fully connected subgraph (we will see why k+1 is required). Then, we transform it into *I(Y)* in poly-time using the following algorithm:

```
DFS_labeling(G,k,prec_male,v):
  counter = number of v.incidentEdges();
  if counter >= k && v.getLabel()==UNEXPLORED:
      if prec_male:
          v.setLabel(F);
          prec_male=0;
      else:
          v.setLabel(M);
          prec_male=1;
  else
      v.setLabel.random(M,F)
  v.setLabel(EXPLORED);
  for all e in v.incidentEdges():
      w = opposite(v,e)
      if w.getLabel() == UNEXPLORED:
          DFS(G,k,prec_male,w);
      else:
          return;
```

This algorithm works because it alternates labels correctly only when an edge has a number of incident edges at least k, using the prec_male boolean, passed from a recursive step to another. In fact, if an edge has less than k incident edges, it will be labeled in a random way and the boolean won't be modified. So, at the end of its execution, every complete subgraph of at least k vertices will be correctly labeled, in an alternate way. This transformed instance *I(Y)* is given as input to our black-box, which returns a graph where |M| = |F| and which contains, for sure, a complete subgraph of at least k vertices since: they are alternate M and F, and the density of k-clique is maximal for each k vertices in it (recall that an edge between two vertices means 1 score, while no edge means 0 score). Then, we retransform this result erasing M and F labels and, as we know that this solution contains a complete graph of dimension at least k (recall that black-box returns optimal solution for our problem Y), we can say that the clique decision problem returns true. *I(X)* contains a complete graph of k+1 instead of k in order to avoid problems with odd numbers: in fact, if k would be an odd number, choosing an *I(X)* containing a k complete graph could lead to problems in the black box: in fact, the output to retransform at the end could be a k-1 clique and our original problem would return false, even if a k-clique was originally present in *I(X)*.

4 Exercise 4

4.1 Problem

In this exercise, we are given a set of tasks presented as a subset S of time istants $\{1..T\}$. Each task can be assigned to a hired employee, and the cost is given by his daily dalary s, or can be outsorced to a freelancer worker, and the cost of this job is c_t. A worker can be hired at any time by paying him a hiring cost C and fired by paying a cost S. We have to:

- Design an algorithm that runs in polynomial time that minimize the total cost of executing the tasks;
- Assuming that now every task needs a number of k workers design an algorithm that runs in polynomial time that minimizes the total cost of executing the tasks.

At any moment there may be any number of tasks, and a worker can handle any number of task at any given time and its cost is still s. We also assume that a task requires one time unit to complete.

4.2 Algorithm 4.1

For the first point, we have to consider that we 'll never pay for a hired employee and a freelancer in the same istant of time. At any given task t_j, considering that the t_{j-1} task has a cost of $Cost_{j-1}$, if we pay both the worker and the freelancer, their cost at the task t_j, will be always bigger than paying for only one, in fact $Cost_{j-1}+C+s+c_j$ is always greater than $Cost_{j-1}+C+s$ or $Cost_{j-1}+C+c_j$. The second observation, which comes from the first, is that if there are tasks within the same time instant it is possible to join them in a single task with the freelancers cost as the sum of the individual freelancer costs of each task. In fact, a worker can work on several tasks together while the same does not apply to freelancers, so the cost of the freelancer is the sum of the costs of the tasks within the same time instant. Joining every subset of overlapping tasks is called, as we decided, *preprocessing step*. In order to define an optimal strategy, we assume to have computed the optimal solution for all the tasks except for the last one, then starting from OPT(j-1) we can have that t_{j-1} task was assigned to an hired employeed h or to a freelancer f. Let's assume that we are on the task t_j and the previous one, t_{j-1} was assigned to a hired employee h. Then, to find the minimum cost for t_j we have two possibilities:

- Current task is assigned to a freelancer, so $OPT(j)=OPT(j-1)+S+c_j$;
- Current task is assigned to the hired employee h, so $OPT(j)=\min\{OPT(j-1)+time\_passed*s, OPT(j-1)+S+C+s\}$ so we have to choose the minimum between continuing to pay the salary of h during the time elapsed between the task t_{j-1} and t_j or to fire and re-hire the employee.

Now let's assume that the previous task t_{j-1} was assigned to a freelancer. Then, the minimum cost can be:

- If current task is assigned to an employee, OPT(j) = OPT(j-1)+C+s;
- If current task is assigned to a freelancer, OPT(j) = OPT(j-1)+c_j.

We provided a Python implementation of this algorithm, attached in the mail. In the code, we used two different arrays for saving costs, one with the initial cost for a freelancer and one with the initial cost for a hired employee. This two arrays are defined as $Cost_s$[] and $Cost_c$[] such that any given task t_j:

- $Cost_s$[j]=min{$Cost_s$[j-1]+time_passed*s, $Cost_s$[j-1]+S+C+s, $Cost_c$[i-1]+C+s}
- $Cost_c$[j]=min{$Cost_c$[j-1]+c_j, $Cost_s$[j-1]+S+c_j}

At the end of the algorithm it will return the minimum value between the last element in $Cost_s$ and $Cost_c$, min{$Cost_s$[T],$Cost_c$[T]}. The preprocessing step has a cost of $O(n^2)$ where n is the number of tasks. The piece of the algorithm where we iterate on tasks costs O(n). We assume that pick an element from the array takes constant time O(1) as well as finding the minimum between two or three numbers. So, the whole algorithm costs $O(n^2)$. The spatial complexity is also linear with the number of tasks since the algorithm uses two arrays for memorization.

4.3 Proof 4.1

We can prove these steps by induction. For the base case we have:

- $Cost_s(0)$=C+s
- $Cost_c(0)$= c_0

From the optimal criteria OPT(j) described before, and by assuming that the $Cost_c$(j-1) and $Cost_c$(j-1) are the lowest possible cost to the task j-1 one endend by an employee h and the other by a freelancer f, then by inductive hypotesis we can say that the optimal ending of task j perfomed by h and the optimal ending of task j perfomed by f are:

- OPT(j)=min{OPT(j-1)+time_passed*s, OPT(j-1)+S+C+s, OPT(j-1)+C+s}= $Cost_s$(j);
- OPT(j)=min{OPT(j-1)+c_j, OPT(j-1)+C+c_j}= $Cost_c$(j);

4.4 Algorithm 4.2 and proof

Differently for the previous case, now every task is composed by k skills and each worker (freelancer and hired employees) has exactly one skill. We assume that every worker has the same salary cost, firing cost and hiring cost. In order to calculate minimum costs for the problem, we will use a list of |T| elements, where T is the tasks subset. This list will contain |T| tuples where the first element of the tuple will contain the optimal permutation with repetition of the j-th task and the second element will contain the minimum cost of this permutation with repetition. For every t_j task we are going to create the matrix of the permutation with repetition of the k skills of t_j, and this matrix has a dimension of 2^k*k because this is the number of permutations of hired employees and freelancers in k positions and every row in the matrix reppresents a permutations with repetition. In the first task t_0 the algorithm will iterate on all the permutations matrix and it will find the row of the matrix with the minimum cost associated in a similar way as we did in the previous paragraph. For each row the cost is calculated as follows: if the element of the row is a worker so if it's 0 the cost will be: cost+=C+s, otherwise the cost will be: cost+= c_t (cost of freelancer for this task). So for the base case: OPT(t_0) = list_opt[t_0][1] and the row associated to this cost is: list_opt[t_0][0]. Suppose that list_opt[t_{j-1}][1] is the minimum cost of the t_{j-1} task (so it is optimal) and list_opt[t_{j-1}][0] is the row associated to this minimum cost. We can find the cost of each row of permutation matrix of t_j task iterating on all the k elements of the row:

- cost += min(list_time[t_j]-list_time[t_{j-1}])*s, S + C + s), if the k-th element is a worker and k-th element of t_{j-1} is a worker;
- cost += C + s, if the k-th element is a worker and k-th element of t_{j-1} is a freelancer;
- cost += S + c_t_cost, if the k-th element is a freelancer and k-th element of t_{j-1} is a worker;
- cost += c_t_cost, if the k-th element is a freelancer and k-th element of t_{j-1} is a freelancer;

After having calculated all the costs of all the rows, the cost of the row with the minimum cost will be the optimum value: OPT(t_j)= min(cost)= list_opt[t_j][1], since the minimum cost found in the task t_j contains the minimum cost of the task t_{j-1} then we can say that the minimum cost of the task t_j is the minimum total cost up to the t_j th task. At the end we can say for sure, when we arrive to analyze the last iteration of permutation of the last task we are sure that list_opt[t_j][1] will be the optimal minimum global cost returned by the algorithm. The cost will be O(kT*2^k), cause we iterate on all tasks and we create all the possible permutations between hired employees and freelancers for each task.

```
list_time <- list of dim |T|
list_opt <- list of |T| tuple

for t in len(task) do
    list_time[t] <- time istant of task[t]
    skills <- skills of task[t]
    c_t_cost <- cost of freelancer of task[t]
    permutation_t <- permutation without repetition of skills (matrix of dim k*2^k)
    min <- MAX_INT

    if list_time[t]==0 then
        for row in permutation_t do
            cost <- 0

            for k in row do
                if k==0 then
                    cost += C + s
                else
                    cost += c_t_cost

            if cost < min then
                min <- cost
                list_opt[t] <- (row,min)
    else
        for row in permutation_t do
            cost <- list_opt[t-1][1]

            for k in len(row) do
                if row[k]==0 then
                    if list_opt[t-1][0][k]==0 then
                        cost+=min(list_time[t]-list_time[t-1])*s,S+C+s)
                    if list_opt[t-1][0][k]==1 then
                        cost+= C + s
                else
                    if list_opt[t-1][0][k]==0 then
                        cost += S + c_t_cost
                    if list_opt[t-1][0][k]==1 then
                        cost += c_t_cost

            if cost < min then
                min <- cost
                list_opt[t]<- (row,min)

return list_opt[t][1]
```

5 Exercise 5

5.1 Problem

In this exercise, we are given a weighted graph $G(V,E)$ and an edge $e \in E$. We have to:

- Design an algorithm which decides whether or not there exists a MST containing e, in time at most $O(|V|+|E|)$
- Design an algorithm which computes a MST containing e, if one exists, in time at most $O(|E|log|E|)$

5.2 Algorithm

For the first point, in order to check if edge e can be contained in a MST, it is sufficient to do a DFS on G considering only edges with weight less than e. We make this DFS start from one of the two vertices connected by edge e. If at the end of the DFS I reached the other vertex connected to e, there is no MST containing e. Otherwise, there is a MST with e. This follows directly from the cycle rule related to graphs, which says: 'Let C be a cycle in a graph. If there is an edge whose weight is the maximum among all edge weights in C, then that edge cannot be part of a MST'. DFS is done in at most $O(|V|+|E|)$, cause we visit each vertex and edge only once. For the second point, as first step we check if a MST containing e exists (using the algorithm explained before), then if returns true, we run Kruskal's algorithm starting from e. The output will be a minimum spanning tree with e, obviously. In the mail containing this PDF I attached a Python version of this algorithm, where I used networkx library in order to create and manipulate graphs. The cost is $O(|E|log|V|)$ in the worst case, cause we used union-find data structure in Kruskal. Doing a better approximation, the number of vertices in a graph is at least $|E|^2$ then: $O(|E|log|V|)=O(|E|log|E|^2)=O(2|E|log|E|)=O(|E|log|E|)$

5.3 Proof of correctness

Proof of correctness for the first point comes immediately from the cycle rule, in fact if I do a DFS from one of the two vertices connected by e, considering only edges with weight less than e, and I reach the other vertex of e, it means that there is a cycle where e is the most expensive edge, so that edge cannot be part of a MST. The second point is a standard use of Kruskal's algorithm, already demonstrated properly in several books related to algorithms and data structures.