Homework 1 - Algorithm Design

Sapienza University of Rome

Marco Costa 1691388

December 7, 2018

1 Exercise 1

The goal of this exercise is to cluster the points of $\mathbf{X}$ using the first $\boldsymbol{k}$ points $\mathbf{C} = \{\pi(\mathbf{X})_\mathbf{1}, ..., \pi(\mathbf{X})_\mathbf{k}\}$ as center (where $\pi(X)$ is a permutation). For any $\boldsymbol{k}$, the output should be optimal with respect of the objective function $\max\limits_{\mathbf{x}\in\mathbf{X}} \min\limits_{\mathbf{c}\in\mathbf{C}} \mathbf{d}(\mathbf{x}, \mathbf{c})$.
This problem is known as k-center clustering problem, and it's NP-HARD. Fortunately, it is possible to find an approximation using a greedy algorithm. The greedy clustering algorithm simply chooses the point farthest away from the current set of centers in each iteration as the new center. It can be described in this way:

- Pick an arbitrary point $\mathbf{c_i}$ into $\mathbf{C}$;
- For every point $\mathbf{x} \in \mathbf{X}$ compute $\mathbf{d_i}(\mathbf{x}, \mathbf{c_i})$;
- Pick the point $\mathbf{c_i}$ with the highest distance from $\mathbf{c_{i-1}}$ ($\max\limits_{x\in X} d(x, c_i)$). In this case, distance must be 3. If there are no points that have the distance 3, then the distance must be 1;
- Add it to the set of centers $\mathbf{C}$;
- Continue till $\boldsymbol{k}$ centers are found (*i=1,...,k*).

1.1 Analysis of the algorithm

- The i^{th} iteration of choosing the i^{th} center takes $\mathbf{O}(|\mathbf{X}|)$ time.
- There are $\boldsymbol{k}$ iterations, so overall the algorithm takes $\mathbf{O}(\mathbf{k}|\mathbf{X}|)$ time.

1.2 Proof

The correctness of this algorithm can be proved by contradiction.
Assume that $\mathbf{O}^* = \{\mathbf{c_1}, ..., \mathbf{c_k}\}$ is an optimal solution for the problem and $\mathbf{S} = \{\mathbf{c_1}, ..., \mathbf{c_k}\}$ the solution given by the greedy algorithm. If $\mathbf{S} \neq \mathbf{O}^*$ so there is at least an optimum center $\bar{\mathbf{c}}$ which belongs to $\mathbf{O}^*$ but not to $\mathbf{S}$. This means that, if this point was selected at the $(\mathbf{i}-\mathbf{1})^{\mathbf{th}}$ iteration of the optimum algorithm because it has distance 3 from the center, so the greedy algorithm has not selected it as the optimum value to satisfy the objective function. But this is not possible, it first insert into the partition all the points of the space which have distance equals to 3 from the center and after the points with distance equals to 1 if needed.

1.3 2-approximation

The solution obtained using the greedy algorithm (the set $\mathbf{C}$) is a 2-approximation to the optimal solution ($\mathbf{r}_\infty^{\mathbf{C}}(\mathbf{X}) \leq \mathbf{2r}_\infty^{\mathbf{opt}}(X, k)$).

1.3.1 Proof

Case 1: Every cluster of $\mathbf{C_{opt}}$ contains exactly one point of $\mathbf{C}$.

- Consider a point $\mathbf{x} \in \mathbf{X}$;
- Let $\bar{\mathbf{c}}$ be the center it belongs to in $\mathbf{C_{opt}}$;
- Let $\bar{\mathbf{k}}$ be the center of $\mathbf{C}$ that is in $\mathbf{\Pi}(\mathbf{C_{opt}}, \bar{\mathbf{c}})$;
- $\mathbf{d}(\mathbf{x}, \bar{\mathbf{c}}) = \mathbf{d}(\mathbf{x}, \mathbf{C_{opt}}) \leq \mathbf{r}_\infty^{\mathbf{opt}}(\mathbf{X}, \mathbf{k})$;
- Similarly, $\mathbf{d}(\bar{\mathbf{k}}, \bar{\mathbf{c}}) = \mathbf{d}(\bar{\mathbf{k}}, \mathbf{C_{opt}}) \leq \mathbf{r}_\infty^{\mathbf{opt}}(\mathbf{X}, \mathbf{k})$;
- By the triangle inequality: $\mathbf{d}(\mathbf{x}, \bar{\mathbf{k}}) \leq \mathbf{d}(\mathbf{x}, \bar{\mathbf{c}}) + \mathbf{d}(\bar{\mathbf{c}}, \bar{\mathbf{k}}) \leq \mathbf{2r}_\infty^{\mathbf{opt}}(\mathbf{X}, \mathbf{k})$.

Case 2: There are two centers $\bar{\mathbf{k}}$ and $\bar{\mathbf{u}}$ of $\mathbf{C}$ that is in $\mathbf{\Pi}(\mathbf{C_{opt}}, \bar{\mathbf{c}})$, for some $\bar{\mathbf{c}} \in \mathbf{C_{opt}}$

- Assume, without loss of generality, that $\bar{\mathbf{u}}$ was added later to the center set $\mathbf{C}$ by the greedy algorithm in the $\mathbf{i}^{\mathbf{th}}$ iteration;
- But since the greedy algorithm always chooses the point furthest away from the current set of centers, we have that $\bar{\mathbf{c}} \in \mathbf{C_{i-1}}$, and $\mathbf{r}_\infty^{\mathbf{C}}(\mathbf{X}) \leq \mathbf{r}_\infty^{\mathbf{C_{i-1}}}(\mathbf{X}) = \mathbf{d}(\bar{\mathbf{u}}, \mathbf{C_{i-1}}) \leq \mathbf{d}(\bar{\mathbf{u}}, \bar{\mathbf{k}}) \leq \mathbf{d}(\bar{\mathbf{u}}, \bar{\mathbf{c}}) + \mathbf{d}(\bar{\mathbf{c}}, \bar{\mathbf{k}}) \leq \mathbf{2r}_\infty^{\mathbf{opt}}(\mathbf{X}, \mathbf{k})$.

2 Exercise 2

It's possible to solve this problem using a bipartite graph $\mathbf{G(S,A,C)}$, where $\mathbf{S}$ is the set containing streets, $\mathbf{A}$ is the set containing avenues and $\mathbf{C}$ is the set containing checkpoints (so a checkpoint is an edge that connects a street in $\mathbf{S}$ with an avenue in $\mathbf{A}$). The goal is to find a vertex set $\mathbf{T} \subseteq \mathbf{S} \bigcup \mathbf{A}$ of minimum size. So the problem is a minimum vertex cover problem (because we want to find a set of vertices such that every edge of the graph has at least one endpoint in the set). It's possible to find $\mathbf{T}$ with:

- Ford-Fulkersons algorithm to find the maximum matching;
- Konig's theorem, which states that in any bipartite graph, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover.

Lets consider $\mathbf{M}$, a maximum cardinality matching in $\mathbf{G}$. A matching is an edge set $\mathbf{M} \subseteq \mathbf{E}$ such that no two edges of M have the same endpoint. Konig's theorem proves that the size of a maximum matching M is equal to the size of the minimum cardinality vertex cover $\mathbf{T}$.
It's possible to find the maximum matching set $\mathbf{M}$ applying Ford-Fulkerson algorithm in the bipartite flow network $\mathbf{G(S,A,C)}$.
Now it's possible to derive from the maximum alternating forest, built starting from non matched vertices in $\mathbf{S}$, the minimum vertex cover set $\mathbf{T}$ using Konig's theorem.
This set represent the minimum set of streets and avenues in which it's necessary to place cameras for covering checkpoints.

2.1 Analysis

The cost of this algorithm is dominated by the finding of the maximum flow (the cost of Ford-Fulkersons algorithm). So the total cost is $\mathbf{O}(\boldsymbol{f}|C|)$, where $\boldsymbol{f}$ is the maximum flow found and $\mathbf{C}$ is the set of the checkpoints.

3 Exercise 3

It's possible to solve this problem considering a complete graph $\mathbf{G(P,R)}$ where $\mathbf{P} = \mathbf{M} \bigcup \mathbf{F}$ is the set of friends ($\mathbf{M}$ is the set of male friends and $\mathbf{F}$ is the set of female friends) and R is the set of their relations (the edges of the graph).
Each edge has weight equals to 0 if the two friends dont like each other, 1

otherwise.
The goal is to find a subset I of people who like each other and who are equally bipartite (so equal number of male and female friends).
The first point of the problem can be solved considering it as a k-clique problem. The algorithm proceed in this way:

1. For each $\boldsymbol{k} \in \{\mathbf{2}, |\mathbf{V}|\}$ it computes the k-clique and stores the cliques with $\boldsymbol{k}$ edges in a set **S**;
2. For each clique in **S** it computes the maximum bipartite matching in order to satisfy the second point (equal number of male and female);
3. For each value of $\boldsymbol{k}$ it finds the maximum bipartite matching $\mathbf{M_k}$ which maximizes the value
$\mathbf{maxVal_k} = \frac{\mathbf{1}}{|\mathbf{M_k}} \sum_{(u,v)\in M_k} weight(u, v)$;
4. At the end it finds the maximum value between the various $\mathbf{maxVal_k}$ with $\boldsymbol{k} \in \{\mathbf{2}, |\mathbf{V}|\}$ and returns the maximum matching corresponding to this value.

3.0.1 Implementation (pseudocode)

```
$V \leftarrow vertices$
$E \leftarrow edges$
$G \leftarrow graph$
$maxD[|V|] \leftarrow [0]$
$maxI[k] \leftarrow [\ ]$
for $k \leftarrow 0$ to $|V|$ do
    $S \leftarrow$ find-all k-cliques()
    for s in S do
        I $\leftarrow$ bipartite matching of s
        if sum weight(s)/$|I| >$ maxD[k] then
            $maxD[k] \leftarrow weight(s)/|I|$
            $maxI[k] \leftarrow I$
$index \leftarrow max(maxD)$
return maxI[index]
```

3.1 Proof

This algorithm uses the k-clique problem, which is NP-COMPLETE, in fact, the reduction of this problem is polynomial. For this reason the implemented algorithm is NP-COMPLETE too.

4 Exercise 4

The input of the algorithm is the following:

- costs: a list of tuple where the first element is the timestamp of the task, the second one is the cost of outsourcing;
- C: hiring cost;
- S: severance cost;
- s: daily salary.

For first, the algorithm computes how many tasks with different timestamp there are. In this way it's possible to overlap tasks. After that, it joins the overlapping tasks generating a matrix in which there are the timestamps in the first row and the outsourcing costs in the second one. Then it generates a total cost matrix and initializes the first column, calculates the lowest cost for each instant using the costs of the previous instant and at the end returns the minimum total cost.

4.1 Analysis

The cost of the algorithm depends on iterative loops so it is linear: O(T), where T is the last time unit to consider.

4.2 Proof

To prove the correctness of the algorithm we could use proof by induction. We consider as basis step the first two couple of values inside total cost matrix , at this moment i = 0 and the min of them is the min cost so far. As induction hypothesis we suppose that for i = k the cost computed as min of the couple of values inside total cost matrix is optimal. The inductive step is to prove that also for i = k + 1 the cost is optimal.

4.3 Implementation

The algorithm is implemented in ***Python***.

Listing 1: minCost

```python
def minCost(costs, C, S, s):

  # Determine the number of different tasks
  T = 0
  time = 0
  for cost in costs:
    if cost[0] != time:
      time = cost[0]
      T += 1

  # Initialize outsourcing cost matrix
  outsource = [[0 for x in range(0, T+1)] for x in range
      (0, 2)]

  # Initialize total cost matrix
  tc = [[0 for x in range(0, T+1)] for x in range(0, 2)]

  # Construck outsourcing cost matrix joining different
      tasks with the same timestamp
  time = 0
  i = 0
  j = 0
  for cost in costs:
    if j == 0:
      outsource[0][i] = cost[0]
      outsource[1][i] = cost[1]
      time = cost[0]
      j += 1
    else:
      if time == cost[0]:
        outsource[1][i] += cost[1]
      else:
        i += 1
        outsource[0][i] = cost[0]
        outsource[1][i] = cost[1]
        time = cost[0]

  # Initialize first column of total cost matrix
  tc[0][0] = s + C
  tc[1][0] = outsource[1][0]

  # Construct rest of the total cost matrix
  for i in range(1, T):
    tc[0][i] = min(tc[0][i-1] + (outsource[0][i] -
        outsource[0][i-1]) * s, tc[1][i-1] + C + s)
    tc[1][i] = min(tc[0][i-1] + S + outsource[1][i], tc
        [1][i-1] + outsource[1][i])

  # Return the minimum cost
  return min(tc[0][T-1], tc[1][T-1])
```

5 Exercise 5

5.1 First part

The first part of the exercise can be solved using the Minimum Spanning Tree cycle property:
*For any cycle **C** in the graph, if the weight of an edge **e** of **C** is larger than the weights of all other edges of **C**, then e cannot belong to the MST.*
The algorithm is based on a **DFS**. It determine if the two endpoints of ***e*** are connected with a cycle respecting the property mentioned before.
How it works:

1. Run the DFS from one of the end-points of the input edge ***e*** (**u** or **v** for the edge ***(u, v, weight)***) considering only outgoing edges which have weight less than the one of ***e***;

2.
   - Case 1: If at the end of the **DFS**, the two vertices **u** and **v** get connected, then the edge ***e*** can't be part of any **MST** because of the cycle property, in fact in this case there exists a cycle in the graph **G** containing the edge ***e*** where ***e*** is the edge with maximum weight.
   - Case 2: If at the end of the **DFS** **u** and **v** stay disconnected, then there is not a cycle between **u** and **v** where ***e*** is the edge with maximum weight, so the edge considered can be part of the **MST**.

The cost of this algorithm is dominated from the one of the **DFS**, which is $\mathbf{O}(|V| + |E|)$.

5.2 Second part

For this second part, the algorithm described before was used to check if the given edge ***e*** could be part of a Minimum Spanning Tree. After checking the previous condition, Kruskals algorithm was used to compute the **MST** containing ***e***: the cost of this algorithm is $\mathbf{O}(|E|log|E|)$.

5.2.1 Implementation

The algorithm is implemented in ***Python***.

Listing 2: MST given an edge

```python
def give_a_mst(self, edge):
  if self.edge_in_mst(edge):
    print("The edge",str(edge),"belongs to MST!")
    print("Minimum Spanning Tree:", end="")
    return self.kruskal()
  else:
    print("The edge",str(edge),"doesn't belong to MST!")
    return None
```

This algorithm is very symple:
It checks if the edge is contained in the MST using the algorithm described in the first part.
If the result is positive, then the Kruskal's algorithm is executed and the MST is output, otherwise nothing is returned.

Listing 3: Kruskal's algorithm

```python
def kruskal(self):
  parent = dict()
  rank = dict()

  for vertex in self.vertex:
    parent[vertex] = vertex
    rank[vertex] = 0

  def find(vertex):
    if parent[vertex] != vertex:
      parent[vertex] = find(parent[vertex])
    return parent[vertex]

  def union(vertex1, vertex2):
    root1 = find(vertex1)
    root2 = find(vertex2)
    if rank[root1] < rank[root2]:
      parent[root1] = root2
    elif rank[root1] > rank[root2]:
      parent[root2] = root1
    else:
      parent[root2] = root1
      rank[root2] += 1

  mst = Graph("directed")
  minimum_spanning_tree = set()
  edges = self.sorted_by_weight()
  for edge in edges:
    v1, v2, w = edge
    if find(v1) != find(v2):
      union(v1, v2)
      minimum_spanning_tree.add(edge)
  for node in self.get_nodes():
    mst.add_node(node)
  for edge in minimum_spanning_tree:
    mst.add_edge(edge)
  return mst
```

A Code

The whole implementation of the exercises are attached in the archive *HW1-AD_Marco_Costa_1691388.tar.gz*