CS 6150: HW 3 – Payman Behnam (U1078370), Greedy Algorithms, Local Search

Submission date: Wednesday, Oct 16, 2019, 11:59 PM

This assignment has 4 questions, for a total of 50 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Set Cover Revisited	16	
Selling Intervals	12	
Forming pairs	10	
Finding diverse elements	12	
Total:	50	

Question 1: Set Cover Revisited . . . . . **[16]**

In class, we saw the set cover problem (phrased as picking the smallest set of people who cover a given set of skills). Formally, we have n people, and each person has a subset of m skills. Let the set of skills of the ith person be denoted by S_i, which is a subset of $[m]$ (shorthand for $\{1, 2, \ldots, m\}$).

(a) **[6]** As before, suppose that there is a set of k people whose skill sets cover all of $[m]$ (i.e., together, they possess all the skills). Now, suppose we run the greedy algorithm discussed in class for $4k$ iterations. Prove that the set of people chosen cover $> 90\%$ of the skills.

Answer: From the class note, we know that $u_{t+1} \leq u_t - \frac{u_t}{k}$. By using plug and chuck we have :

$$u_{t+1} \leq u_t(1 - \frac{1}{k}) \leq u_{t-1}(1 - \frac{1}{k})^2 \leq \ldots\ldots\ldots \leq u_0(1 - \frac{1}{k})^t$$

Overall, after *t* iteration, for the uncovered set we have: $u_t \leq m(1 - \frac{1}{k})^t$. After t=4k, we have $u_t \leq m(1 - \frac{1}{k})^{4k}$ and we know $(1 - \frac{1}{k})^k = \frac{1}{e}$. Hence, $u_t \leq m(\frac{1}{e})^4 = m \times 0.0183$. So, number of covered skills are: $m - 0.0183m = 0.98m$ which is more than $> 90\%$ of the skills.

(b) **[4]** Consider the following "street surveillance" problem. We have a graph (V, E) with n nodes and m edges. We are allowed to place surveillance cameras at the nodes. Once placed, they can monitor all the edges incident to the node. The goal is to place as few cameras as possible, so as to monitor **all the edges** in the graph. Show how to cast the street surveillance problem as Set Cover.

Answer: We can map edges to streets and they can be considered as skills in Set Cover problem. The nodes are mapped to cameras which are people in the Set Cover problem. The question in Set Cover was this: Suppose we have n people, and m "desired skills"; each person has a subset of the skills. Pick the smallest number of people such that every skill is covered. In our case, we are looking for smallest number of camera(nodes, people) such hat every street(edges, skills) is covered.

(c) **[6]** Let (V, E) be a graph as above, and suppose that the optimal solution for the street surveillance problem places k cameras (and is able to monitor all edges). Now consider the following "lazy" algorithm:

1. initialize $S = \emptyset$
2. while there is an unmonitored edge $\{i, j\}$:
   add both i, j to S and mark all their edges as monitored

Clearly (due to the while loop), the algorithm returns a set S that monitors all the edges. Prove that the set also satisfies $|S| \leq 2k$ (recall that k is the number of nodes in the optimal solution).

MORAL. Even though the algorithm looks "dumber" than the greedy algorithm, it has a better approximation guarantee — 2 versus $\log n$.

Hint. Consider the edges $\{i, j\}$ encountered when we run the algorithm. Could it be that the optimal set chooses *neither* of $\{i, j\}$?

Answer: Suppose that we run an optimal algorithm and we encounter the edge $\{i, j\}$ which is not covered already. In the optimal algorithm, any way we need to put a camera on a specific node (e.g. *i*) to cover the edge (sometimes we need to add camera on both nodes). There isn't any better solution since the street needs to be monitored and we need to put a camera on a node such that it can be monitored. This is not the case that we can ignore adding cameras on at least one of the $\{i, j\}$ .That camera covers all the edges connected to the node *i*. However, In the proposed lazy algorithm, when we encounter to edge $\{q, t\}$, we add both q, t to S and mark all their edges as monitored. Instead of that in the optimal solution we need to put at least one camera. This means that the number of selected cameras in the proposed lazy algorithm in the worst case is 2 times larger than the optimal solution (it can be less than that of course since the optmum solution may put two camera for one selected edge). In other words $|S| \leq 2k$. Please not the "optimal algorithm" encounters (i,j) pairs not necessarily in the same as the proposed lazy algorithm encounters.

Question 2: Selling Intervals . . . . . **[12]**

Suppose we are given n intervals on the real line — $[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]$. Of course, some of the intervals overlap. For each interval, we have a buyer, who is willing to pay a price of p_i for interval i. The goal is to find the best subset of intervals to sell, such that (a) intervals sold are all non-overlapping, and (b) the total price is maximized.

(a) **[4]** It is natural to want to sell intervals that are short but have a high price. Consider a greedy algorithm that first sorts the intervals in decreasing order based of the ratio of price to length, i.e., $p_i/(b_i - a_i)$, and then does the following:

1. initialize $S = \emptyset$
2. go over the intervals in sorted order (as above), doing the foll:
   add the interval to S if it does not overlap with any interval already in S

Give an instance in which the greedy choice is sub-optimal.

Answer: Suppose that we have the following intervals and prices (i.e., (interval, price))

L1: ([1,3], 8), L2: ([4,6], 8), L3: ([15,16], 1)

L4: ([2,5], 9)

L5: ([2,14], 24)

According to the algorithm we have the following ratios for L1 to L5 respectively: 4,4,1,3,2. So the sorted ratio would be $\{L1, L2, L4, L5, L3\}$. Considering the above algorithm, we should have only L1 and L2, L3 and the total price is 8+8+1 =17. However, we can pick only L5 and L3 and the price would be 24+1 =25.

(b) **[8]** Design an algorithm based on dynamic programming. Your algorithm should run in time polynomial in n. As always, include an analysis of correctness and the running time.

Suppose that p(j) = the largest $i < j$ such that interval *i* doesn't overlap with *j*. In this case, p(j) is the interval farthest to the right that is compatible with j.

The question is what does an optimum solution look like? Let OPT be an optimal solution. and *n* be the last interval. If OPT solution contains interval *n*, OPT = $n + Optimalsolutionon\{1, ..., p(n)\}$. Otherwise, for OPT we need to find optimal solution on $\{1, ..., n-1\}$.

Lets suppose that OPT(j) is the optimal solution for only intervals 1, . . . , j. Based on the above intuition, OPT(j) is the maximum of the three following values:

$$\begin{cases} vj + OPT(p(j)) & \text{(if j is in OPT solution)} \\ OPT(j-1) & \text{(if j is not in OPT solution)} \\ 0 & \text{(j = 0)} \end{cases}$$

If we write a recursive algorithm based on the above formula, its running time would be exponential. The reason is that we have to compute a lot of repetitive examples (e.g., like Fibonacci sequence.) To avoid this problem we can rely on dynamic programming procedure (Memoize) and save the answer for each subproblem when we compute it and reuse it in the future.

Algorithm 1 Interval Scheduling Algorithm1

```
1: procedure INTERVALSCHEDULING1(j)
2:     if j = 0 then
3:         Return 0
4:     else if M[j] is not empty then
5:         Return M[j]
6:     else
7:         M[j] = max(v[j] +IntervalScheduling(p[j]), IntervalScheduling(j-1))
8:         Return M[j]
```

We need to Fill in 1 array entry for every two calls, and hence the complexity would be O(n)

However, when we compute M[j], we only need values for M[k] for k ¡ j and hence we can have:

Algorithm 2 Interval Scheduling Algorithm2

```
1: procedure INTERVALSCHEDULING2
2:     Input: n, a1,...,an , b1,...,bn , v1,...,vn
3:     Sort intervals by finish times (bi) so that b1 ≤ b2 ≤ ... ≤ bn
4:     Compute p(1), p(2), ..., p(n)
5:     M[0] = 0
6:     j = 1
7:     while j ≤ n do
8:         M[j] = max(v[j] + M[p(j)], M[j-1])
```

Proof of correctness:

Lets consider the optimal solution M[j]. Either interval *j* is employed or not. If not, then we have a maximum weight set of non-overlapping intervals from 1 to *j* which is M[j - 1]. If interval *j* is used for M[j], since intervals p(j) + 1 through *j* - 1 can have all overlap with interval *j*, this means that the remaining interval selected for M[i] are drawn from 1 through p(j). If we eliminate *j* from the optimal solution for M[j], it gives us non overlapping intervals on intervals 1 to p(j). If $M[p(j)]$ is an optimal solution, $M[j] - v_j \leq M[p(j)]$. However, if we add interval *j* to M[p(j)] – which is possible – it only makes use of intervals up through *j*. Accordingly $M[j] - v_j \geq M[p(j)]$. So, we came to the conclusion that $M[j] = v_j + M[p(j)]$. We are looking for maximization. Hence, the max of $(v[j] + M[p(j)], M[j-1])$ is the right answer.

Example:

The following example (Fig 1) shows how we can find maximum non-overlapping interval values through dynamic programming and fill the array. Red ones are selected ones by the algorithm.

To print the selected intervals we can use the following procedure:

Interval n is included in the optimal solution if $v[n] + M[p[n]] \geq M[n-1]$. After deciding if n is in the solution, we can look at the subproblem $\{1, 2, ..., n-1\}$. So the algorithm would be:

Algorithm 3 Print Intervals

```
1: procedure PRINTINTERVALS(M, j)
2:     if j = 0 then
3:         print empty
4:     if j > 0 then
5:         if v[j] + M[p[j]] ≥ M[j − 1] then
6:             print j
7:             PrintIntervals(M,P[j])
8:         else
9:             PrintIntervals(M,j-1)
```

Example:

The following example (Fig 2) shows how we can print the selected intervals . The oval shows selected intervals from previous example.

Running Time complexity:

Time to sort by finishing time: $O(nlogn)$ Time to compute p(n): $O(n^2)$ Time to fill in the M array: $O(n)$ Time to backtrack and print solution: $O(n)$

My references are : https://www.cs.cmu.edu/ ckingsf/bioinfo-lectures/dynamicp.pdf

http://www.cs.cornell.edu/courses/cs482/2007su/dynamic.pdf

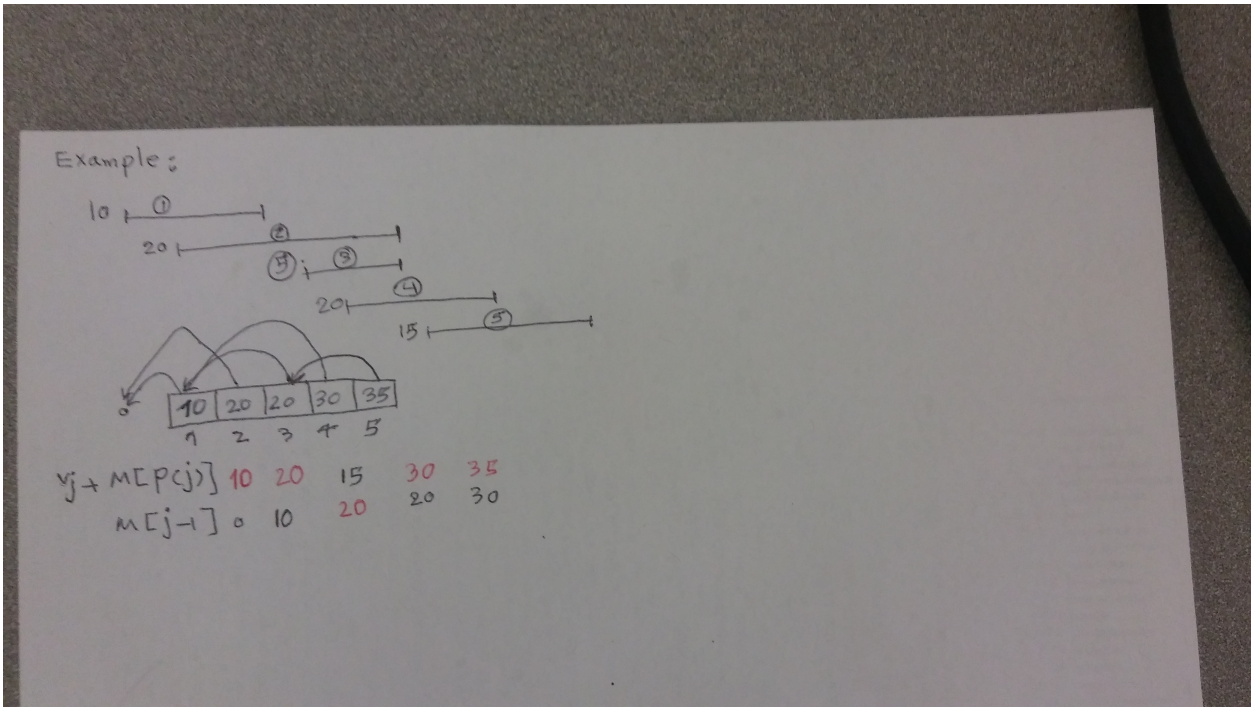

Figure 1: find maximum non-overlapping interval value

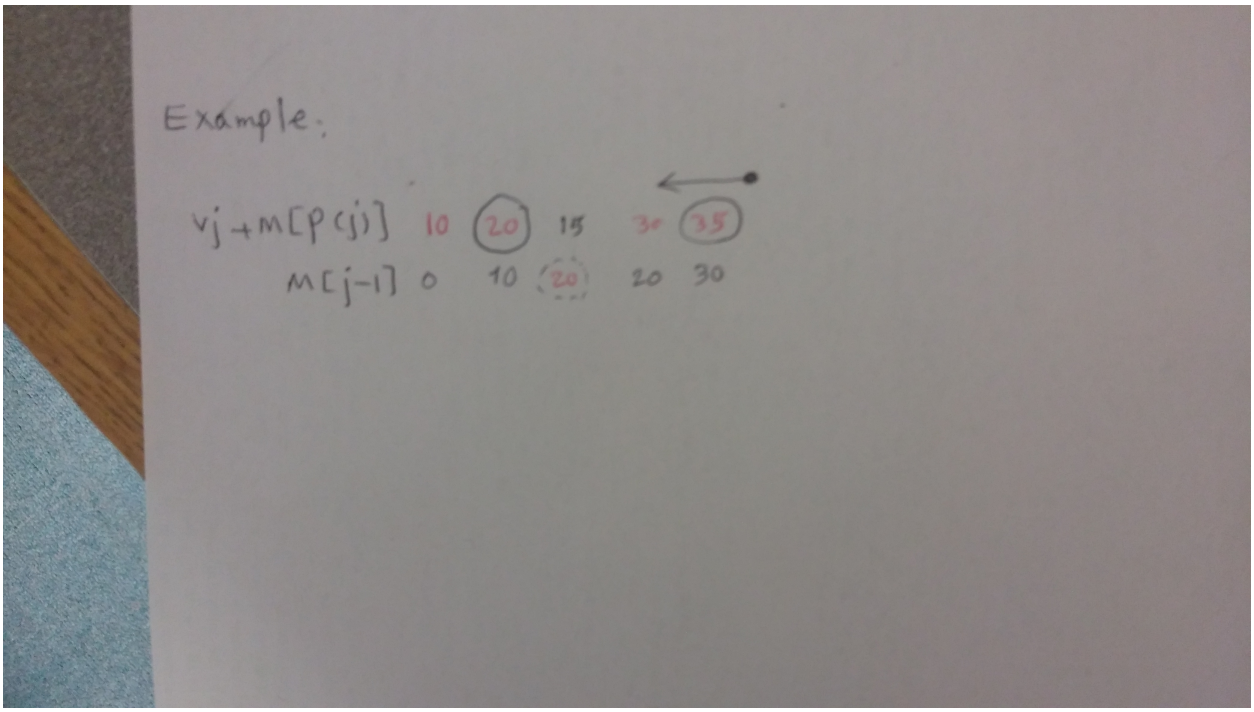

Figure 2: print the selected intervals

Question 3: Forming pairs . . . . . **[10]**

Consider n people, each having a *skill level* ($s_1, s_2, \ldots, s_n$, which are integers). We are also given a threshold T. Two people form a *successful pair* if the sum of their skill levels is $\geq T$.

The goal is to form the maximum number of successful pairs. For convenience, suppose that s_i are already sorted in increasing order. Now, consider the following greedy algorithm: given a set of people, consider pairing the least skilled person and the most skilled person. If the sum of the skill levels is $\geq T$, form the pair and recurse; else, delete the least skilled person and recurse on the rest.

(a) **[3]** Show how to implement this algorithm in $O(n)$ time (by writing pseudo-code and performing a runtime analysis).

Algorithm 4 PairFinding-Recursive

```
1:  procedure PAIRFINDING(s[n], start, end)
2:      p= {}
3:      i=start; j=end
4:      if (i >= j) then
5:          return P
6:      else if (s[i] + s[j] >= T) then
7:          add (s[i], s[j]) to P;
8:          i++
9:          j- -
10:         PairFinding(s[n], i, j)
11:     else
12:         i++
13:         PairFinding(s[n], i, j)
```

In the recursive function, we are moving from the left and right sides towards right and left sides. Calling functions takes O(1) and we are calling them N times. So, the complexity would be O(N).

Algorithm 5 PairFinding-NonRecursive

```
1:  procedure PAIRFINDING(s[n], start, end)
2:      p= {}
3:      i=start; j=end
4:      while i < j do
5:          if (s[i] + s[j] >= T) then
6:              add (s[i], s[j]) to P
7:              i++;
8:              j- -
9:              PairFinding(s[n], i, j)
10:         else
11:             i++
12:             PairFinding(s[n], i, j)
```

In the Non-recursive function, we have while loop that iterates N times. Each element in while loop is viewed 2 times in the worst case and this happens N times. The complexity would be O(N).

(b) **[7]** Does the algorithm always output the pairing with the largest possible number of (successful) pairs? Either provide a counter-example, or give a formal proof of correctness.

Answer The algorithm is optimal. Let's suppose that the algorithm doesn't output the pairing with the largest possible number of (successful) pairs. What does it mean? It means that there is at least one pair that is included in the optimal solution set, but our algorithm was not able to capture it. Now, the question is how this pair (i.e.,

(s_i, s_j)) looks like. I will follow why at least one of s_i or s_j has not been matched with any and as a result they are matched together.

Suppose in our algorithm, s_i hasn't been matched with s_t while $t > j$: Either s_t already has been matched with s_d while $d < i$. If this is the case, it doesn't matter either we put (s_i, s_t) or (s_d, s_t). Since we are not looking for a unique solution but in the largest set it doesn't matter which one we pick. Or s_t cannot be matched with (i.e., s_i) and other members before s_i while $t > j$. In other words, $\forall e \leq s_i : si + st < T$. If this is the case, how come $s_i + s_j > T$ while $t > j$? So, it is not possible! Or s_t has been matched with s_k while $i < K < j$. In this case, since our algorithm traverse the array from left to right (and right to left) and $i < K$, it has already tried s_i and the matching condition was failed $s_i + s_t < T$. So, no point to to have neither (s_i, s_t), nor (s_i, s_j). Or $s_i + s_j > T$, s_i and s_j are not matched with any other previous or afterward members and the algorithm didn't capture it. It is not possible. Algorithm traverse the array from left to right (and right to left) in order. In each step, after changing pointers, for any uncovered pair if $s_i + s_j > T$, the algorithm will capture the pair. So, the conclusion is that if we select k intervals, but optimum solution gives $k + 1$ intervals we have contradiction. In other words, the output of the proposed algorithm is as good as the optimum algorithm.

Question 4: Finding diverse elements ..... **[12]**

A common problem in returning search results is to display results that are *diverse*. A simplified formulation of the problem is as follows. We have n points in Euclidean space of d-dimensions, and suppose that by distance, we mean the standard Euclidean distance. The goal is to pick a subset of k (out of the n) points, so as to maximize the sum of the pairwise distances between the chosen points. I.e., if the points are denoted $P = \{p_1, p_2, \ldots, p_n\}$, then we wish to choose an $S \subseteq P$, such that $|S| = k$, and $\sum_{p_i, p_j \in P} d(p_i, p_j)$ is maximized.

A common heuristic for this problem is local search. Start with some subset of the points, call them $S = \{q_1, q_2, \ldots, q_k\} \subseteq P$. At each step, we check if replacing one of the q_i with a point in $P \setminus S$ improves the objective value. If so, we perform the swap, and continue doing so as long as the objective improves. The procedure stops when no improvement (of this form) is possible. Suppose the algorithm ends with $S = \{q_1, \ldots, q_k\}$. We wish to compare the objective value of this solution with the optimum one. Let $\{x_1, x_2, \ldots, x_k\}$ be the optimum subset.

(a) **[3]** Use local optimality to argue that:

$$d(x_1, q_2) + d(x_1, q_3) + \cdots + d(x_1, q_k) \leq d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k).$$

Answer: We know the algorithm ends up with S. This means that swapping one of them with other point should not increase the sum of pairwise distances between the members of S (due to the local search property that was performed). For instance, if we swap the q_1 with x_1, still sum of pairwise distances between the set $\{x1, q2, q3, .., qk\}$ should be less than or equal of sum of pairwise distances of S= $\{q1, q2, q3, ..., qk\}$ with k points. Therefore, for sum of pairwise distances we have:

$$d(x_1, q_2) + d(x_1, q_3) + \cdots + d(x_1, q_k) + \sum_{i,j>2, i,j \in S} d(q_i, q_j) \leq$$

$$d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k) + \sum_{i,j>2, i,j \in S} d(q_i, q_j)$$

by canceling sigma in both side we have:

$$d(x_1, q_2) + d(x_1, q_3) + \cdots + d(x_1, q_k) \leq d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k).$$

(b) **[4]** Deduce that: [*Hint:* Use two inequalities of the form above.]

$$(k-1) \cdot d(x_1, x_2) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$

Answer: Based on the first proven part we have:

$$d(x_1, q_2) + d(x_1, q_3) + \cdots + d(x_1, q_k) \leq d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k).$$

suppose that in the first part instead of swapping q_1 with x_1 we swap it with x_2. So we have:

$$d(x_2, q_2) + d(x_2, q_3) + \cdots + d(x_2, q_k) \leq d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k).$$

If we add these two inequality we have:

$$d(x_1, q_2) + d(x_1, q_3) + \cdots + d(x_1, q_k) + d(x_2, q_2) + d(x_2, q_3) + \cdots + d(x_2, q_k) \leq$$
$$2[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)]$$

Based on triangle inequality of Euclidean distance we know that

$$d(x_1, q_i) + d(x_2, q_i) \geq d(x_1, x_2)$$

If we substitute triangle inequality in the extracted equation we have:

$$d(x_1, x_2) + d(x_1, x_2) + \cdots + d(x_1, x_2) \leq$$

$$d(x_1, q_2) + d(x_1, q_3) + \cdots + d(x_1, q_k) + d(x_2, q_2) + d(x_2, q_3) + \cdots + d(x_2, q_k) \leq$$
$$2[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k).]$$

since we have k-1 element (e,g., indexes cannot be equal):

$$(k-1) \cdot d(x_1, x_2) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$

(c) **[5]** Use this expression to argue that

$$\sum_{i,j} d(x_i, x_j) \leq 2 \sum_{i,j} d(q_i, q_j).$$

Note that this shows that the locally optimum solution has objective value at least $1/2$ the optimum.
Answer: From the part (b) we know that :

$$(k-1) \cdot d(x_1, x_2) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$

But x_2 can be replaced with any other x_j. The reason is that we are relying on the triangle inequalities.

$$d(x_1, q_i) + d(x_2, q_i) \geq d(x_1, x_2)$$

For instance we can have:

$$d(x_1, q_i) + d(x_3, q_i) \geq d(x_1, x_3)$$

or

$$d(x_1, q_i) + d(x_4, q_i) \geq d(x_1, x_4)$$

and so on so forth. As a result of that we can have:

$$(k-1) \cdot d(x_1, x_2) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$
$$(k-1) \cdot d(x_1, x_3) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$
$$(k-1) \cdot d(x_1, x_4) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$

. . .

$$(k-1) \cdot d(x_1, x_k) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$

If we add them up we have:

$$(k-1)\sum_j d(x_1, x_j) \leq 2(k-1)\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$

(k-1) can cancel out each other. So we have:

$$\sum_j d(x_1, x_j) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$

Similarly, By relying on the triangle inequalities we know that

$$d(x_1, q_2) + d(x_2, q_2) \geq d(x_1, x_2)$$

$$d(x_3, q_3) + d(x_2, q_3) \geq d(x_3, x_2)$$

$$d(x_4, q_4) + d(x_2, q_4) \geq d(x_4, x_2)$$

Hence, in the last extracted equation we can replace x_1 with x_2, x_3,.... and q_1 with q_2, q_3, ... We will have the following

$$\sum_j d(x_1, x_j) \leq 2\left[d(q_1, q_2) + d(q_1, q_3) + \cdots + d(q_1, q_k)\right].$$

$$\sum_j d(x_2, x_j) \leq 2\left[d(q_2, q_1) + d(q_2, q_3) + \cdots + d(q_2, q_k)\right].$$

$$\sum_j d(x_3, x_j) \leq 2\left[d(q_3, q_1) + d(q_3, q_2) + \cdots + d(q_3, q_k)\right].$$

. . .

$$\sum_j d(x_k, x_j) \leq 2\left[d(q_k, q_1) + d(q_k, q_2) + \cdots + d(q_k, q_{k-1})\right].$$

If we add them up we will have the following equation :

$$\sum_i \sum_j d(x_i, x_j) \leq 2 \sum_{i,j} d(q_i, q_j).$$

So we have:

$$\sum_{i,j} d(x_i, x_j) \leq 2 \sum_{i,j} d(q_i, q_j).$$

Which is the equation that we were asked to prove.