

Local Search Heuristics for k -median

Lecturer: David Kempe

Scribe: Kaveh Shahabi, Rumi Ghosh, Shuo Liu and Congkai Sun

1 Problem Definition

In the k -median and facility location problems, we are given two sets: F , the set of *facilities* and C , the set of *clients*. There is a specified *distance* $c_{ij} \geq 0$ (forming a metric) between every pair $i, j \in F \cup C$. The goal is to identify a subset of facilities $S \subseteq F$ and to serve the clients in C by the facilities in S , such that the cost function is minimized. Formally, the k -median problem is defined as:

Definition 1 Select S with $|S| = k$, minimizing $\text{cost}(S) = \sum_{\text{clients } j} c_{\sigma(j),j}$ where $\sigma(j)$ is the center that j is assigned to (and thus in $\text{argmin}_{i \in S} c_{ij}$).

The difference between k -median and facility location is that in the facility location problem, each facility has a cost (and the algorithm may choose the number of facilities to open), while in k -median, the number of facilities that can be opened is given as an input, and cannot be exceeded.

We will analyze a simple local search algorithm for k -median, due to Arya, Garg, Khandekar, Meyerson, Munagala, and Pandit. While local search heuristics are frequently used in practice (because of their simplicity), it is not too often that one can prove good approximation guarantees for them. This is one of the few cases. (Some others include a $\frac{1}{2}$ -approximation for Max-Cut and a 2-approximation for Image Segmentation.) In general, local search algorithms have the following outline:

- Start with some solution.
- We have a set of small moves with which we can update the solution.
- In each step, find a move that (most) improves the solution over the move space.
- Continue to do this until we get a local extremum.
- Output the local extremum.

The hope is that the local extremum is not too much worse than the global extremum. This clearly depends on how we define the move set. The richer the set of moves, the better the local extremum tends to be; but in return, in each iteration, we have to try more choices. In the extreme case, we could allow exhaustive search over all solutions as a “local move”, which clearly will find the optimum, albeit in exponential time.

For the k -median algorithm, the local moves are very simple: swap one facility for another. Formally, the algorithm is as follows:

Algorithm 1 The Local Search Algorithm for k -median

Start with an arbitrary set S of size $|S| = k$.

while there is an $i \in S, i' \notin S$ with $\text{cost}(S \cup \{i'\} \setminus \{i\}) < \text{cost}(S)$ **do**

Let $S = S \cup \{i'\} \setminus \{i\}$.

The algorithm is under-specified: if there are multiple i, i' whose swaps decreases the cost, it does not say which swap to perform. In practice, a good heuristic is to choose the one minimizing $\text{cost}(S \cup \{i'\} \setminus \{i\})$. While it would converge faster in practice, it does not help with worst-case analysis.

First, the result of this algorithm is clearly feasible, since the only condition is having at most k facilities selected, which is explicitly maintained.

Next, we want to analyze termination and running time. Notice that in each iteration, the cost strictly decreases, and there are at most $\binom{|F|}{k}$ candidate sets of k facilities. Because the cost strictly decreases, the same set can never be revisited, so it takes at most $\binom{|F|}{k}$ iterations. The running time may, however, be pseudo-polynomial in the costs (assuming they are integer). We will revisit in Section 3 how to make the algorithm polynomial.

2 Approximation Guarantee

Let S be the solution of the algorithm and S^* the optimum solution. The algorithm's termination guarantees that

$$\text{cost}(S \cup \{i'\} \setminus \{i\}) \geq \text{cost}(S) \quad \text{for all } i, i'. \quad (1)$$

We will use this inequality for $i' \in S^*, i \in S$, to argue that the optimum solution cannot be too much better (as adding it in small increments does not help us). From now on, for emphasis, we use $o \in S^*$ to denote optimum facilities and $s \in S$ to denote the algorithm's facilities. It is possible that some facilities occur in both sets S and S^* . Similarly, we use $C(j)$ for the cost of client j in the algorithm's solution, $C^*(j)$ for the cost of client j in the optimum solution, and $N(i), N^*(i)$ for the set of clients assigned to $i \in S$ and $i \in S^*$, respectively.

Our overall approach is to define a set of swaps $\langle o, s \rangle$, and use the Inequality 1 to bound the increase in cost $D_{o,s} = \text{cost}(S \cup \{o\} \setminus \{s\}) - \text{cost}(S) \geq 0$. We will ensure that each $o \in S^*$ appears in exactly one pair, and each $s \in S$ appears in at most two pairs. When adding up these costs for each pair $\langle o, s \rangle$, we will obtain the optimum cost, compared with a constant times the algorithm's cost.

The important question is then which s should be paired with a given o . To this end, we first define the notion of "capturing". We say that $s \in S$ captures $o \in S^*$ iff $|N(s) \cap N^*(o)| > \frac{1}{2}|N^*(o)|$. That is, s in S serves more than half the clients o serves in S^* . Intuitively, that makes s a good candidate to be swapped with o . Problems might occur, though, when s captures many different o . We therefore define $s \in S$ as *bad* iff it captures more than one $o \in S^*$ (see Figure 1).

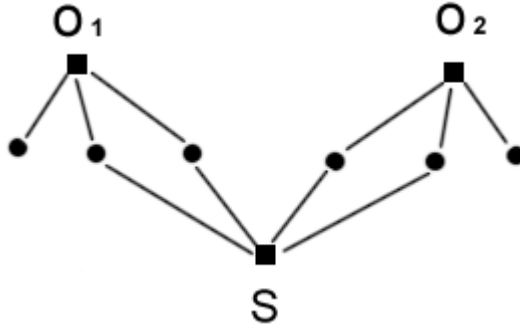


Figure 1: s is bad since it is capturing more than one o

We will later see that for our proof, it does not work when a bad facility s is used to pair with any o . The reason, intuitively, is that when s is swapped out for o , then the clients that cannot be safely reassigned to o will not have a natural place to be reassigned to. So for the pairings, we will never use bad facilities s . The strategy is as follows:

1. If o is captured by s and s is not bad, then use the pair $\langle o, s \rangle$.
2. Pair all the remaining $o \in S^*$ arbitrarily with the remaining $s \in S$ that are not bad, such that no s is paired more than twice.

We need to argue that the second case is in fact possible. Let U be the set of optimum facilities remaining unpaired after the first step, and G be the set of facilities that are not bad and remain unpaired after step 1. We need that $|G| \geq \frac{1}{2}|U|$; then, any way of pairing them up will work. This is true because each bad s captures at least two $o \in U$, and each $o \in U$ is captured at most once. Thus, the number of bad $s \in S$ is at most $\frac{|U|}{2}$. So $|G| \geq \frac{|U|}{2}$.

Next, we want to use all the versions of Equation 1 ($\text{cost}(S \cup \{o\} \setminus \{s\}) - \text{cost}(S) \geq 0$) to get an upper bound on $\text{cost}(S^*)$. To do this, the crucial step is obtaining a good upper bound on $\text{cost}(S \cup \{o\} \setminus \{s\})$ which will connect the cost to $\text{cost}(S^*)$. To do this, we define a reassignment of clients j after a swap, since some of the clients are going to lose their centers.

One part of the reassignment is fairly straightforward: all $j \in N^*(o)$ are reassigned to o . This will introduce the optimum cost once we sum over all pairs $\langle o, s \rangle$. But we still have to deal with the clients $j \in N(s) \setminus N^*(o)$. For these clients, we define a “backup plan” permutation π , which maps each client j to some client $j' = \pi(j)$; then, j will use the facility currently used by j' to replace s when needed. To be useful, this permutation should be such that $\pi(j)$ is “fairly close” to j . At the same time, it should be a permutation: if we mapped multiple clients to the same j' , then accounting for the costs would not work out.

It turns out that the following property of π is what we need: For every $o \in S^*$, and every $s \in S$ such that $|N(s) \cap N^*(o)| \leq \frac{1}{2}|N^*(o)|$, all $j \in N(s) \cap N^*(o)$ satisfy $\sigma(\pi(j)) \neq \sigma(j)$. In other words, $\pi(j)$ uses a different facility than j . If $|N(s) \cap N^*(o)| > \frac{1}{2}|N^*(o)|$ (i.e., s captures o), then $\pi(j)$ can be arbitrary for all $j \in N(s) \cap N^*(o)$.

First, we need to ensure that such an assignment exists. One way to generate π is to number all clients $j \in N^*(o)$ such that the $N(s) \cap N^*(o)$ form contiguous blocks. Then, we can map j to $(j + \frac{1}{2}|N^*(o)|) \bmod |N^*(o)|$. (See Figure 2). It is now straightforward that this definition satisfies the condition we need.

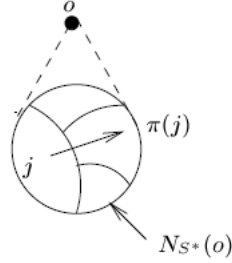


Figure 2: Finding the reassignment. (Figure taken from the paper by Arya et al.)

Having defined π , we now formally state that the new facility for every $j \in N(s) \setminus N^*(o)$ is $\sigma(\pi(j))$. First, we need to ensure that the resulting reassignment is feasible. If s captures some o' , then because only good s are swapped out, the swap pair was actually $\langle o, s \rangle$, and all $N(s) \cap N^*(o)$ get reassigned to o (which is open). If s does not capture any o' , then by the property of π , $\sigma(\pi(j)) \neq \sigma(j) = s$, so the new facility assigned to j is also open (see Figure 3).

To analyze the cost of the swap, we can write

$$\begin{aligned}
0 &\leq \text{cost}(S \cup \{o\} \setminus \{s\}) - \text{cost}(S) \\
&\leq \sum_{j \in N^*(o)} (C^*(j) - C(j)) + \sum_{j \in N(s), j \notin N^*(o)} (c_{\sigma(\pi(j)), j} - C(j)) \\
&\leq \sum_{j \in N^*(o)} (C^*(j) - C(j)) + \sum_{j \in N(s), j \notin N^*(o)} (C^*(j) + C^*(\pi(j)) + C(\pi(j)) - C(j))
\end{aligned}$$

To obtain the final inequality, we used the triangle inequality, as illustrated in Figure 4:

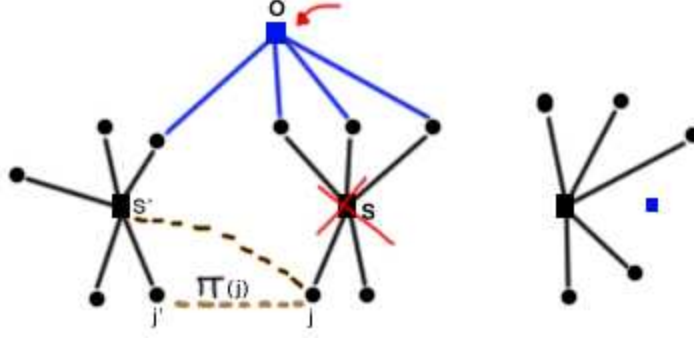


Figure 3: s is swapped out and o is swapped in. j is being reassigned.

$$c_{\sigma(\pi(j),j)} \leq c_{(o',j)} + c_{(o',j')} + c_{(s',j')}$$

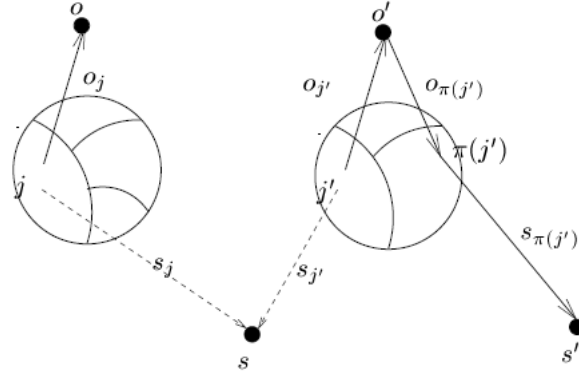


Figure 4: Reassignment and the triangle inequality. (Figure taken from the paper by Arya et al.)

Now, summing over all matched pairs $\langle o, s \rangle$:

$$\begin{aligned}
0 &\leq \sum_{\langle o, s \rangle} \left(\sum_{j \in N^*(o)} (C^*(j) - C(j)) + \sum_{j \in N(s), j \notin N^*(o)} (C^*(j) + C^*(\pi(j)) + C(\pi(j)) - C(j)) \right) \\
&\leq \sum_o \sum_{j \in N^*(o)} (C^*(j) - C(j)) + 2 \sum_s \sum_{j \in N(s)} (C^*(j) + C^*(\pi(j)) + C(\pi(j)) - C(j)) \\
&= \text{cost}(S^*) - \text{cost}(S) + 2(\text{cost}(S^*) + \text{cost}(S^*) + \text{cost}(S) - \text{cost}(S)) \\
&= 5 \cdot \text{cost}(S^*) - \text{cost}(S).
\end{aligned}$$

In the second inequality, we used that the value under the second sum was non-negative, and that each $s \in S$ was swapped out at most twice. In the next equality, we used that π was a permutation, so summing over all j or over all $\pi(j)$ produces a sum over all clients. Finally, by rearranging the last inequality, we obtain that $\text{cost}(S) \leq 5 \cdot \text{cost}(S^*)$, and thus a 5-approximation.

The above algorithm can be generalized to swap in and out pairs or triples or — generally — p -tuples of facilities simultaneously. Notice that this increases the move space size to $\Theta(n^{2p})$. A similar, but slightly

more technical, analysis shows that this gives a $(3 + 2/p)$ -approximation. This bound is tight in the worst case, as can be seen from an example in the paper by Arya et al.

3 Making the Algorithm Polynomial-Time

Many local search algorithms in their basic form are pseudo-polynomial. While the objective function decreases strictly in each iteration, it may only decrease by small amounts. The following trick frequently works in those cases to make the running time polynomial at a cost of a small ϵ in the approximation guarantee.

Instead of iterating until no more improvement is possible, we only iterate until *no big improvement* is possible, where “big” is defined in terms of an ϵ and a polynomial scaling factor. In the special case of the k -median problem with $p = 1$, the new algorithm is:

Algorithm 2 Polynomial Local Search Algorithm for k -median

Start with an arbitrary set S of size $|S| = k$.

while there is an $i \in S, i' \notin S$ with $\text{cost}(S \cup \{i'\} \setminus \{i\}) \leq (1 - \delta) \cdot \text{cost}(S)$ **do**

Let $S = S \cup \{i'\} \setminus \{i\}$.

We will set the value of δ momentarily. The termination condition of the algorithm now guarantees that

$$\text{cost}(S \cup \{o\} \setminus \{s\}) - \text{cost}(S) \geq -\delta \cdot \text{cost}(S) \tag{2}$$

instead of the previous

$$\text{cost}(S \cup \{o\} \setminus \{s\}) - \text{cost}(S) \geq 0.$$

We still choose the pairs $\langle o, s \rangle$ as before. When we sum up all the inequalities 2, we now obtain that $5 \cdot \text{cost}(S^*) - \text{cost}(S) \geq -\delta k \cdot \text{cost}(S)$, or $\text{cost}(S) \leq \frac{5}{1-\delta k} \text{cost}(S^*)$.

Since we want that $\frac{5}{1-\delta k} = 5 + \epsilon$, we can solve to obtain $\delta = \frac{\epsilon}{k(5+\epsilon)} = \Theta(\frac{\epsilon}{k})$. So in order to get a $(5 + \epsilon)$ -approximation, we just need to set $\delta = \frac{\epsilon}{5k}$.

The impact on the running time is as follows: Each iteration of the local search takes $O(kn)$ time, so the crucial part is calculating the number of iterations the algorithm will take. Assume that all C_{ij} are integers. The initial cost C_0 is at most $\sum_{i,j} C_{ij}$. In each round, $\text{cost}(S)$ decreases by at least a factor $1 - \delta$. Once $\text{cost}(S)$ is strictly less than 1, termination is guaranteed. So the total number of iterations is at most

$$\log_{\frac{1}{1-\delta}} C_0 = \frac{\ln C_0}{-\ln(1-\frac{\epsilon}{5k})} = \frac{\ln C_0}{\ln(5k) - \ln(5k-\epsilon)} \leq \frac{\ln C_0}{\frac{1}{5k} \epsilon} = \Theta\left(\frac{k \ln C_0}{\epsilon}\right).$$

The inequality followed from a derivative bound. Thus, the running time is polynomial (though not strongly polynomial).