Summer Report On The Capacitated K-Center Problem

Anya Chaturvedi K. R. Prajwal
Ketaki Vaidya Sagar Sahni

July 4, 2016

1 Introduction to our Problem

The capacitated K-center problem is basically a facility location problem, where one is asked to locate K facilities in a graph and to assign vertices to these facilities. In doing this we minimize the maximum distance from a vertex to the facility to which it is assigned while keeping in mind that each facility may be assigned at most L vertices including itself. This problem is known to be NP-hard.

We show our attempts at solving this problem, counter examples as learnt by us and few basic implementations of the same. The implementations include finding the optimal using the integer program and a partially accurate local search algorithm.

2 A Look At The K-center problem

The problem taken up by us is actually a generalization of the K-center problem which we will explain here. Given n vertices with specified distances, one wants to build k facilities at different vertices and minimize the maximum distance of a vertex to a facility. This problem is NP-hard. An approximation algorithm with a factor of η ,for a minimization problem,is a polynomial time algorithm that guarantees a solution with cost at most η times the cost of an optimal solution. For the basic K-center methods have been presented for obtaining an approximation factor of 2. Given a complete undirected graph G = (V, E) with distances $d(v_i, v_j) \in N$ satisfying the triangle inequality, find a subset $S \subseteq V$ with $|S| = k$.

Input:
1. The vertices must be in a metric space, or in other words a complete graph that satisfies the triangle inequality.
2. An upper bound on the number of centers K.

Formal Definition:

$$\min_{S \subseteq V} \max_{v \in V} \min_{s \in S} d(v, s)$$

where d is the distance function.

This is much similar to the problem of placing k disks such that all points are covered in the set V and thus finding the minimum radius for the disks.

3 The Capacitated K-Center

The capacitated K-center problem is nothing but a generalization of the K-center problem. We have to output a set of at most K centers,as well as an assignment of vertices to centers. No more than L vertices may be assigned to a single center. Under these constraints,we wish to minimize the maximum distance between a vertex u and its assigned center φ(u).

Input:
1. The vertices must be in a metric space, or in other words a complete graph that satisfies the triangle inequality.
2. An upper bound on the number of centers K.
3. A maximum load L.

Formal Definition:

$$\min_{S \subseteq V} \max_{u \in V} d(u, \varphi(u))$$

such that,

$$|\{u|\varphi(u) = v\}| \leq L \forall v \in S,$$

where,

$$\varphi : V \to S.$$

The first polynomial time approximation algorithm for this problem was with an approximation factor of 10. Later a 6-approximation was found which is the best up till date while a 5-approximation works when we can assign multiple centers at a single vertex without including the vertex in L. This was our problem for the summer.

Initially we went through the 2.6 and 9.3 sections of Williamson Schmoy which solve the min degree spanning tree problem which is also NP-hard and got to know of methods like local search used to create approximation algorithms.

Certifying a graph for an algorithm not only has to ensure that we can show whether a solution is possible for G or not it also involves showing that if a solution is not possible in G then it is not even possible in $G^2, G^3 \ldots$

4 Checking the Feasibility of the Input Graph

We say that, if we solve the capacitated K-center problem then it will be equivalent to getting the minimum value of W for which G(W) has a capacitated dominating set of size k, where G(W) refers to the graph having all edges with edge weight less than or equal to W.

We can easily set up such an equivalence if we slightly modify our problem. Given a graph $G = (V, E)$ and a capacity function $c : V \to N$, $S \subseteq V$ is a capacitated

dominating set if there exists a mapping $f : V \backslash S \rightarrow S$ (domination mapping) such that:

1. $\forall v \in V \backslash S, \forall s \in S, f(v) = s \Longrightarrow (v, s) \in E$

2. $\forall s \in S, |\{v \in V \backslash S | f(v) = s\}| \leq c(s)$

In the new version of the k-center problem, the goal is to get a set S of k vertices and an assignment $h : V \backslash S \rightarrow S$ of every vertex (in $V \backslash S$) to an open center such that the longest distance between a vertex and a center it is assigned to is minimum and no facility is assigned more vertices than its capacity. Note that, the assignment is not for all vertices in V . We are making the assumption that every center serves itself and the capacity (L(v)) of a vertex $v \in V$ is the maximum number of clients it can serve (excluding itself).

Now we can say that, solving the capacitated k-center problem is equivalent to getting the minimum value of W for which G(W) has a capacitated dominating set of size k. $S \subseteq V$ is an optimal solution to the capacitated K-center problem if and only if w(S) is the minimum value of W for which the graph G(W) has a capacitated dominating set of size k. It is also easy to verify that a capacitated dominating set of size k in $G(w(S))^i$, for some i and some optimal solution S, forms an i-approximation for the k-center problem.

But the capacitated dominating set problem is an NP-complete problem. So, we cannot get a capacitated dominating set of size k in polynomial time, unless $P = NP$ Similar to the uncapacitated version the capacitated version too is hard to approximate within a factor of 2.

4.1 Verification using a Network Flow

Given a graph $G = (V, E)$ and a capacity function $c : V \rightarrow N$, checking whether $S \in V$ is a capacitated dominating set can be viewed as a network problem. Construct a directed graph $G_S = (V_S, E_S)$ as follows:

1. $V_S = V \cup \{s, t\}$

2. $E_S = \{(s, v_i) | v_i \in V \backslash S\} \cup \{(s_i, t) | s_i \in S\} \cup \{(v_i, s_i) | v_i \in V \backslash S \wedge s_i \in S \wedge (v_i, s_i) \in E\}$

Let $c_S : E_S \rightarrow N$ be the capacity function such that,

$$c_S(e_{uv}) = \begin{cases} (c(u), & v = t \\ 1, & otherwise \end{cases}$$

In the final graph G_S we add two more vertices s (source) and t (sink) to the vertex set of G. We add directed edges from s to all vertices in $V \backslash S$, from each vertex in $V \backslash S$ to its neighbours in S and each vertex in S to t. We take the capacity function to be equal to the capacity of the source vertex for edges incident on t and to be equal to 1 for every other edge in E_S.

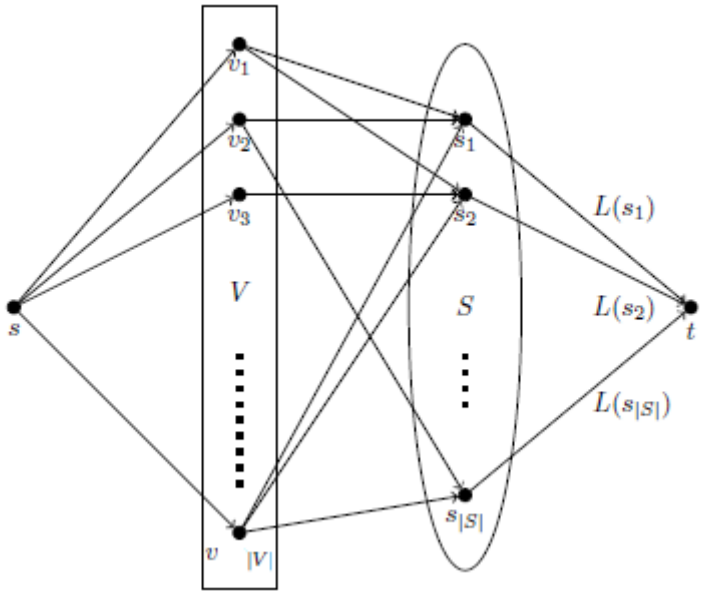

Figure 1: The flow network

Now it is easy to verify that, S is a capacitated dominating set in G if and only if G_S has a maximum coverage of $|V \backslash S|$ from s to t equal to the total vertices in the graph.

5 The Integer Program

Linear programming is a mathematical technique for maximizing or minimizing a linear function of several variables such as output or cost. The decision variables with respect to which the optimum is obtained can come out to be fractional or any real number. An integer programming problem is a mathematical optimization or feasibility program in which some or all of the variables are restricted to be integers which is the major difference it has from linear programming. Ours is a problem where we wish to minimize the maximum distance between a vertex u and its assigned center thus our integer program is framed as:

```
import snap
from pulp import *

def randomGraph(n,edges):
        UGraph = snap.GenRndGnm(snap.PUNGraph, n, edges)
        adj = dict()
        for x in range(n):
                for y in range(n):
                        adj[(x,y)] = 0
        for EI in UGraph.Edges():
                adj[(EI.GetSrcNId(),EI.GetDstNId())] = 1
                adj[(EI.GetDstNId(), EI.GetSrcNId())] = 1
        for i in xrange(n):
                adj[(i,i)] = 1
        return adj

N=input("No. of vertices (n<4039):")
edges=input("No. of edges:")
adj = randomGraph(N,edges)
edges = sum([sum(x) for x in adj])/2
L = N/10

# declare your variables
cen = dict(zip([i for i in range(N)],[0 for i in range(N)]))
cen = LpVariable.dicts("cen",cen, 0, 1,LpInteger)

```

Figure 2: Code

```
assignments = {}
for i in range(N):
        for j in range(N):
                assignments[(i,j)] = adj[(i,j)]

assignments = LpVariable.dicts("assignments",assignments,0,1)

# defines the problem
prob = LpProblem("problem", LpMinimize)

# defines the objective function to minimize : sum(y)
prob += lpSum([cen[i] for i in range(N)])

############# Constraint definitions ###################

# sum(x_ij) <= L*y[i]
for i in xrange(N):
        prob += lpSum([assignments[(i,j)] for j in xrange(N)]) <= L*cen[i]

# sum(x_ij) = 1 for all j
for j in xrange(N):
        prob += lpSum([assignments[(i,j)] for i in xrange(N)]) == 1

# y_i >= x_ij for all i,j
for i in xrange(N):
        for j in xrange(N):
                prob += assignments[(i,j)] <= cen[i]
#the assignment can be done only when the edge is present
for i in xrange(N):
        for j in xrange(N):
                prob += assignments[(i,j)] <= adj[(i,j)]

# solve the problem using GLPK
status = prob.solve(GLPK(msg=0))
print LpStatus[prob.status]
list_var=prob.variables()
list_cen=list_var[(len(list_var)-N):]
count=0
for v in list_cen:
    if(v.varValue==1.0):
        count+= 1
print N,edges,count
```

Figure 3: Code

Objective Function:

$$z = \sum_{i=1}^{N} cen[i]$$

Variables:

1. ***cen*** - an integer dictionary which specifies the centers among the vertices where if $cen[i] = 1$ then the i^{th} index vertex is taken to be a center otherwise not.
2. ***assignments*** - dictionary $assignments[i][j] = 1$ will represent that the j^th index vertex is assigned to the center created at the i^th index vertex, we initialize it with the adjacency matrix
3. ***N*** - represents the total number of vertices
4. ***adj*** - a dictionary which stores the adjacency matrix of the graph

Constraints:

1. $\forall i \sum_{j=1}^{N} assignments[i][j] \leq L * cen[i]$
   ...makes sure no more than L vertices are assigned to a center
2. $\forall j \sum_{i=1}^{N} assignments[i][j] = 1$
   ...a vertex can be assigned only to a single center
3. $\forall i \forall j\, assignments[i][j] \leq cen[i]$
   ...a vertex can only be assigned to a second vertex if the second vertex is a center
4. $\forall i, \forall j\, assignments[i][j] \leq adj[i][j]$
   ...assignment of vertices can only be done if the edge is present in the original graph

6 Local Search Algorithm

Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. A local search algorithm starts from a candidate solution and then iteratively moves to a neighbour solution. We move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution optimal is found. We have used a local search algorithm which was formed by Aounon. It doesnt work on a set of graphs but we get the solution for some general graphs we tested on.

Here, we start with a set S of random k vertices in graph G . Let S be the set of all k vertex sets which are formed by replacing one vertex in S by a vertex in V (one-swap vertex sets). We look for the set in S which maximizes the number of dominated vertices. If the optimum set dominates more vertices than S then we set S to that set and continue the process till we have reached a local optima. We do the assignments through a network flow graph.

Algorithm:

$S \leftarrow v1, v2, \ldots, vk$
$S' \leftarrow one - swap\ vertex\ sets\ of\ S$
while $MaxFlow(G_{V_{max}}, c_{V_{max}}) > MaxFlow(G_S, c_S)$ **do**
    $S \leftarrow V_{max}$

$S' \leftarrow one - swap vertex sets of S$
$V_{max} \leftarrow argmax_{V \in S} MaxFlow(G_V, c_V)$
end while
if $MaxFlow(G_S, c_S) = n - k$ **then**
for all $v \in V$ **do**
if $f_{max}(v, s) = 1$ **then**
$h(v) = s$
end if
end for
return S, h
end if

```
import networkx as nx
import snap
from random import shuffle
from collections import defaultdict

#Generating a random graph using snap
def randomGraph(n,edges):
        UGraph = snap.GenRndGnm(snap.PUNGraph, n, edges)
        adj = dict()
        for x in range(n):
                for y in range(n):
                        adj[(x,y)] = 0
```

Figure 4: Code

```
        for EI in UGraph.Edges():
                adj[(EI.GetSrcNId(),EI.GetDstNId())] = 1
                adj[(EI.GetDstNId(), EI.GetSrcNId())] = 1
        for i in xrange(n):
                adj[(i,i)] = 1
        for x in range(n):
                for y in range(n):
                        print adj[(x,y)],
                print
        return adj

#Storing the random graph in G
def getGraph(N,adj):
        G = nx.Graph()
        G.add_nodes_from([x for x in xrange(N)])
        for i in xrange(N):
                for j in xrange(i,N):
                        if adj[(i,j)]:
                                G.add_edge(i,j)
        return G

#Obtaining a random set S of initial k centers
def getS(G,k):
        nodes = G.nodes()
        shuffle(nodes)
        return nodes[:k]

#making FG a directed flowgraph to give as input to max flow functions ( s-->V-->S-->t )
def getFlowGraph(G, S, L):
        FG = nx.DiGraph()
        N = len(G.nodes())
        #for common nodes in V and S make separate keys for S
        for x in G.nodes():
                FG.add_node(x)
                if x in S:
                        FG.add_node(x + N)

        FG.add_nodes_from(['s','t'])
        edges = set(G.edges())
        for s in S:
                for v in G.nodes():
                        if (min(s,v),max(s,v)) in edges:
                                FG.add_edge(v, s + N, capacity=1)
        for v in G.nodes():
                FG.add_edge('s',v,capacity=1)
```

Figure 5: Code

```
        for s in S:
                FG.add_edge(s + N,'t',capacity=L)
        return FG

#Obtaining the set Vmax with one swap from S to get the max flow value set
def getVmax(G, S, L):
        V = G.nodes()
        # flow[i][j] = max-flow in (S, V) when S[i] is swapped with V[j]
        flow = [[None for _ in xrange(len(V))] for _ in xrange(len(S))]
        setS = set(S)
        for i in xrange(len(S)):
                for j in xrange(len(V)):
                        if V[j] not in setS:
                                temp = S[i]
                                S[i] = V[j]
                                flow_graph = getFlowGraph(G, S, L)
                                flow[i][j] = nx.maximum_flow_value(flow_graph, 's', 't')
                                S[i] = temp

        max_value = 0
        swap_pair = None
        for i in xrange(len(S)):
                for j in xrange(len(V)):
                        if max_value < flow[i][j]:
                                swap_pair = (i,j)
                                max_value = flow[i][j]
        i,j = swap_pair
        S[i] = V[j]
        return S

#changing S and Vmax by one swaps so as to achieve the max flow possible after each obtained S
def doOneSwaps(G, S, L):
        Vmax = getVmax(G, S, L)
        while nx.maximum_flow_value(getFlowGraph(G, Vmax, L), 's', 't') > nx.maximum_flow_value(getFlowGraph(G
                S = Vmax
                Vmax = getVmax(G, S, L)
        return S

N, M = 10,30
adj = randomGraph(N,M)
k, L = 5,3
G = getGraph(N,adj)
```

Figure 6: Code

```
S = getS(G,k)
S = doOneSwaps(G, S, L)
H = defaultdict(list)
#max flow value cannot be greater than the number of nodes in the graph or set V
if nx.maximum_flow_value(getFlowGraph(G, S, L), 's', 't') == N:
        _,flows = nx.maximum_flow(getFlowGraph(G, S, L), 's', 't')
        edges = set(G.edges())
        #if a flow exists along a particular path assigning v to that vertex in S
        for v in G.nodes():
                for s in S:
                        if (min(v,s),max(v,s)) in edges:
                                unitflow = False
                                try:
                                        if flows[max(v,s + N)][min(v,s + N)] == 1:
                                                unitflow = True
                                except KeyError:
                                        try:
                                                if flows[min(v,s + N)][max(v,s + N)] == 1:
                                                        unitflow = True
                                        except KeyError:
                                                pass
                                if unitflow:
                                        H[s].append(v)
else:
        print "Failed!"
print S
print H
```

Figure 7: Code

7 Star Graph Generation

In an attempt to try our programs on a more clustered data set, this program was implemented such that in this star graphs are formed which are sparsely connected to each other by edges which are added to the graph based on a probability p.

```
import snap
import random
import networkx as nx
from collections import namedtuple

class Graph(object):
        def __init__(self, specs):
                self.k = specs.k
                self.l = specs.l

                if specs.struct == 'stars':
                        self.N = self.k*self.l
                        self.p = specs.p# supplementary edge probability
                        self.starGraph()

                elif specs.struct == 'random':
                        self.N = specs.N
                        self.edges = specs.M
                        self.randomGraph()

                else:
                        raise ValueError('Invalid specifications!')

                self.nxGraph = self.NXify()

        def initAdj(self):
                self.adj = dict()
                for x in range(self.N):
                        for y in range(self.N):
                                self.adj[(x,y)] = 0
                for i in xrange(self.N):
                        self.adj[(i,i)] = 1

```

Figure 8: Code

```
34          def randomGraph(self):
35                  UGraph = snap.GenRndGnm(snap.PUNGraph, self.N, self.edges)
36                  self.initAdj()
37
38                  for EI in UGraph.Edges():
39                          self.adj[(EI.GetSrcNId(),EI.GetDstNId())] = 1
40                          self.adj[(EI.GetDstNId(), EI.GetSrcNId())] = 1
41
42          def starGraph(self):
43                  self.initAdj()
44                  non_centers = set([x for x in xrange(self.N)])
45                  star_centers = random.sample(list(non_centers), self.k)
46                  for x in star_centers: non_centers.remove(x)
47                  non_centers = list(non_centers)
48                  random.shuffle(non_centers)
49                  star_clients = [non_centers[i:i + self.l - 1] for i in xrange(0, len(non_centers),
50
51                  for center, clients in zip(star_centers, star_clients):
52                          for client in clients:
53                                  self.adj[(center,client)] = self.adj[(client, center)] = 1
54
55                  for x in range(self.N):
56                          for y in range(x + 1,self.N):
57                                  if self.adj[(x,y)] != 1:
58                                          self.adj[(x,y)] = int(random.random() < self.p)
59                                          self.adj[(y,x)] = int(random.random() < self.p)
60
61
62          #Storing the random graph in G
63          def NXify(self):
64                  G = nx.Graph()
65                  G.add_nodes_from([x for x in xrange(self.N)])
66                  for i in xrange(self.N):
67                          for j in xrange(i,self.N):
68                                  if self.adj[(i,j)]:
69                                          G.add_edge(i,j)
70                  return G
```

Figure 9: Code

8 Observations made

This section basically focuses on all our trials and errors we had during our intern on this topic along with few of the concepts learnt which provide the base for these trials.

8.1 Lower Bound on K

Lets say we make some partition on our input graph such that each partition has two types of vertices. Inner and boundary, the inner vertices are the ones within the particular partition while the boundary vertices are inner vertices which even have edges

to other partitions. Let the i^th partition be represented by its inner vertice C_i and its boundary vertices as a function of C_i as $B(C_i)$.

Since we know K vertices can cover up to L vertices so the max vertices than can be covered is $V = K * L$. Hence K can be taken to be approximately $\lceil V/L \rceil$. Thus if we try forming a lower bound so as to claim that the graph requires atleast these many vertices as equal to the bound then it might be of help.

Here, in the partitioning as described above, for the vertices other than the boundary vertices we can say that they can only be covered by vertices of their own partition thus we need to open centers in the partition itself for such vertices. Thus, if we say there are p partitions then $\sum_{i=1}^{p} \lceil (C_i - B(C_i))/L \rceil$ vertices are required atleast to cover the graph and hence this comes to be a lower bound.

This has been presented in a research paper already but the fact that we are not really involving the boundary vertices did not sink in and though it did not work out but we toyed for some time with ideas of somehow involving the boundary vertices. We thought with the current vertices the remaining capacity i.e. $\lceil (C_i - B(C_i))/L \rceil * L - (C_i - B(C_i))$ we may be able to cover some or even all of the boundary vertices. Through this we can decide opening more centers or shifting the centers so as to cover more boundary vertice. But it always depends upon the connectivity of the particular graph and what kind of partition we take and that was where we left this.
Later when Jatin and Aounon started thinking towards this way we further applied our thinking on the same.

Lets say we partition the graph into sets of nodes C_i $1 \leq i \leq N$: As discussed before, the minimum number of centers we need to place inside the partition C_i is $\lceil ((C_i - B(C_i))/L) \rceil$. The residual capacity, which can cover boundary vertices as mentioned above is,$\lceil (C_i - B(C_i))/L \rceil * L - (C_i - B(C_i))$, which we can use to cover the boundary vertices.

To do this, a flow network is created:
Now there are two cases:

- If we are not able to find a flow to cover all the boundary vertices with the residual capacities, then there exist a set of partitions $P_1, P_2, \ldots P_i$ and $B(P_1), B(P_2), \ldots B(P_i)$ for which Hall's theorem doesnt hold i.e.:

$$|B| > \sum_{i=1}^{N} \lceil (P_i - B(P_i))/L \rceil * L - (P_i - B(P_i))$$

  which after rearrangement becomes:

$$\lceil (\sum_{i=1}^{N} P_i)/L \rceil > \sum_{i=1}^{N} \lceil P_i/L \rceil$$

  This in turn means we can merge the above set of partitions to get a better lower bound. The boundary vertices form the interior of the new set of partitions.

- If the flow exists, this is where we are stuck again. We do not know if there is such a set of centers we can choose such that we can make use of enough residual

flow required and actually taken into account by the flow network. Our attempt to resolve this: We define

- Node 'a' : A center which does not have edge to any boundary vertex.
- Node 'b' : An interior non - center, which is a client of 'a' and adjacent to a boundary vertex. We will call it the semi boundary vertex.

We observe that if we make as many semi-boundary centers as possible, such that they cover maximum number of boundary vertices, by switching the centers from 'a' to 'b' then we can come to a conclusion. If we make the semi-boundary vertex 'b' as a center, then it will cover all the clients that have been covered by 'a', but its minimum maximum distance would be 4*OPT. Also, we would need to shift to such a set of semi-boundary vertices that maximises the number of distinct boundary vertices getting covered.

So, we make a flow network: That is, we cannot make greedy moves and just move to the semi-boundary vertices with maximum boundary neighbors. As it is equally necessary to make sure these boundary neighbours are distinct. To ensure this, we create the following flow network: But we realised that this problem of covering maximum number of boundary vertices will eventually be comparable to the max-coverage problem, which is NP-hard.The comparison is made by taking the nodes inside the partition as the subsets and the universe outside. If there is a way where we can get minimum subsets to cover all of the universe then it is the same solution. Since it is NP-hard we know this won't be possible.

However, we still feel that this idea of semi-boundary vertices could lead to some result.

8.2 Our Attempt

- In our attempt we start by applying the 2-approximation K-center algorithm on the given graph. We then tried to apply what we had learnt by reading the OPT + 1 local search algorithm of the min-degree spanning tree from section 2.6 of Williamson & Schmoy. Effectively,if possible we transferred the load from the highest capacity nodes to lower ones through a contiguous sequence of edge assignment swaps until we reach a vertex with lower capacity as visible in the figure.

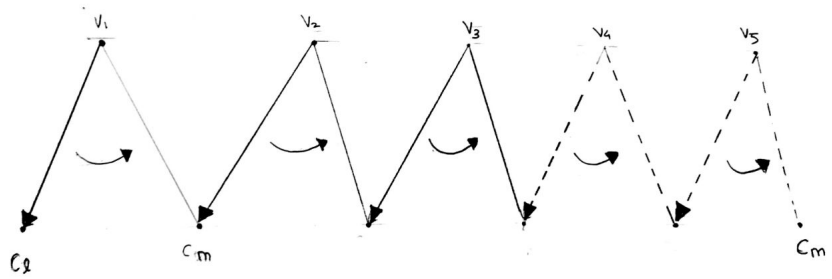

Figure 10: reassignment through propagation

Algorithm:

1: Start with k' centers of G^2 with no regard to capacity L using the 2-approximation algorithm for uncapacitated k-center.
2: $C \leftarrow$ set of all k-centers
3: $C^l \leftarrow$ subset of C such that$\forall C \in C^l$, C covers exactly l clients.
4: In each phase:
5: $l \leftarrow$ the maximum degree of any center
6: $C_l \leftarrow$ set of all centers with degree l
7: Subphase
8: **while** there are center with degree l **do**
9: Choose a client v assigned to some center in C_l.
10: Choose another client C_m (m¡l) of minimum degree among all the centers v is connected to or some C_m which is connected through some alternate manner of center then non center and finally with v.
11: **end while**

This is done in order to reduce the maximum number of assignments to a particular center so as to reach the capacity L. This propagation is possible only when a non center is connected to two or more centers and hence reassignment can be done.

But we still might have too few centers in dense cliques while just sufficient in other parts or maybe even having residual capacities in sparse areas. We also have a counter example for the same as shown in the figure below. The centers are represented by the rectangular marks on the vertices. The left most center is covering many clients within a distance of ¡=2 where as the same cannot be said for the other centers.

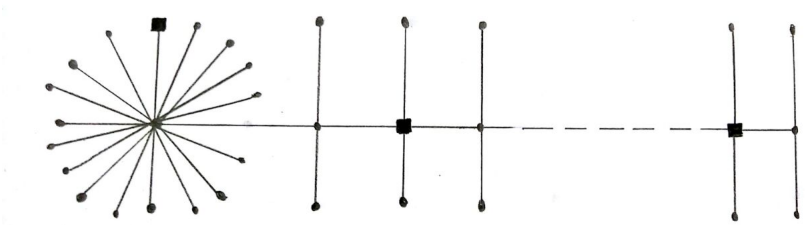

Figure 11: Counter Example

Maybe moving to higher powers of G may contribute to requiring less centers at regions where at present the centers are suffice.

Hence, from G^2 we move to G^4. Here, each center with its clients forming a star graph will transform into a clique.The same shown in the figure below.

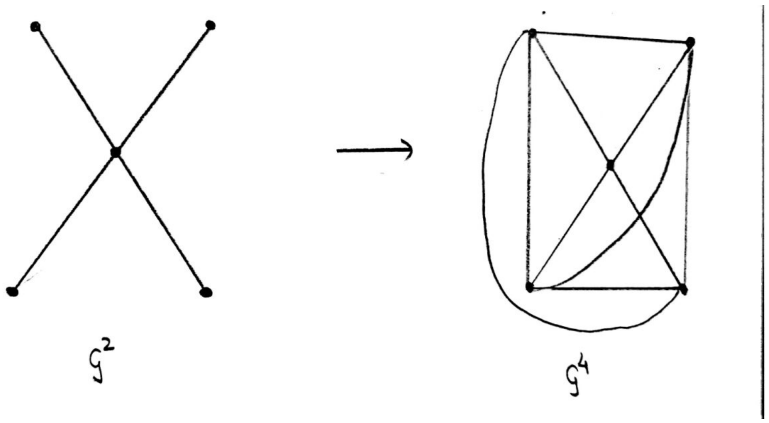

Figure 12: Clique formation

With such a change we might be able to merge few cliques by placing a center on a common node of two cliques(while removing their respective centers) such that both the cliques add up to give vertices less than L.

Following this we have free centers available which we can place in dense cliques with high degree of vertices. Like in the figure below in the nodes that were centers in G^2 are represented by square labels in the left figure and diamond label on the right while the one which we replace them by in G^4 is represent by the square in the right figure.

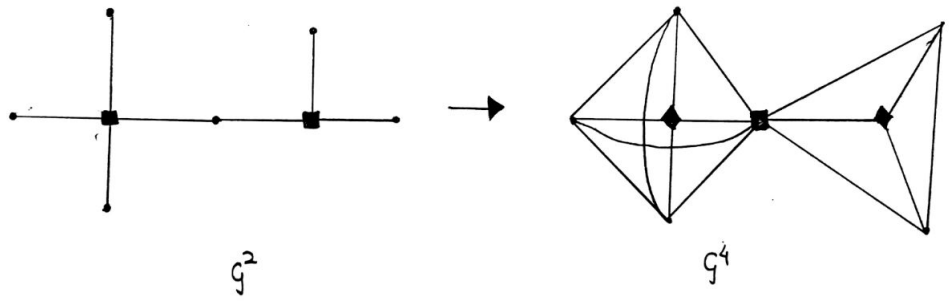

Figure 13: Merging

Like here we obtain one extra center. The problem still remains on how to keep obtaining such centers other than by merging and then placing them at an appropriate position.

Further, we try to come up with some kind of data structure that might help us make some claims on the lower bound on K. As above we make a level structure with the cliques in set C_l as taken before at level l.After a reassignment is performed, the structure changes, because, always the cliques are taken to be a center and its neighbors. After a reassignment, the set of neighbors of (more than or equal to 2) centers change, and also, their capacities change, thus moving them to different levels than present.

Theorem: When there can be no more edge reassignments, then no center at level l, has an edge of length ≤ 4 to any node in level $l' \geq l + 2$ for all l.

Proof: After there can be no more reassignments, let there exists such a center C at a level l which has an edge to some node P belonging to some level l' ¿= l + 2. Let C' be the center covering the node P. Then we can do a reassignment step such that P gets assigned to C instead of C' thereby moving C' one level down and C one level up. Thus the max degree can decrease this way removing one clique at a time.

This contradicts our assumption that no more reassignment is possible.

8.3 Counter Examples

8.4 Current Scenario