

The Capacitated K-Center Problem

A Summer Internship Report (9 May-8 July)

Anya Chaturvedi
Sagar Sahni

K. R. Prajwal
Ketaki Vaidya

July 6, 2016

Contents

1 Abstract

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. Starting by describing our problem we go through the flow method which helps us verify our solution to the problem. Next we also did implementations of our problem's integer program and a partially correct local search algorithm. The local search algorithm's implementation gives us a very high accuracy though there are definitely some counter examples for it which we have visualised and have given here. In implementing the above programs we have used three ways of making the input graphs. One is a random graph, second a more clustered form i.e. a star graph and the last one is generated by taking points in a unit square. Other than this we have included our observations which maybe not much conclusive but are our trials on solving the problem by the approaches we learnt during our intern here.

2 A Look At The problem

The problem taken up by us is actually a generalization of the K-center problem which we will explain here.

2.1 The K-Center Problem

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.

2.2 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 $\varphi(u)$.

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,

$$\varphi : V \rightarrow S.$$

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 Schmoys 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 $G^2, G^3 \dots$

3 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 $c : V \rightarrow N$, $S \subseteq V$ is a capacitated dominating set if there exists a mapping $f : V \setminus S \rightarrow S$ (domination mapping) such that:

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

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

In the new version of the k-center problem, the goal is to get a set S of k vertices and an assignment $h : V \setminus S \rightarrow S$ of every vertex (in $V \setminus S$) 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.

3.1 Verification using a Network Flow

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

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

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

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

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

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 $V \setminus S$, from each vertex in $V \setminus S$ 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 .

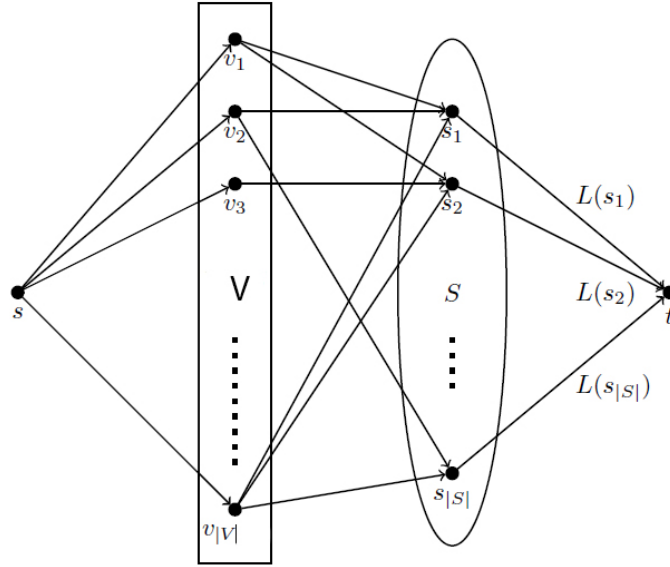


Figure 1: The flow network

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 $|V \setminus S|$ from s to t equal to the total vertices in the graph.

4 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:

```

1  import snap
2  from pulp import *
3
4  def randomGraph(n,edges):
5      UGraph = snap.GenRndGnm(snap.PUNGraph, n, edges)
6      adj = dict()
7      for x in range(n):
8          for y in range(n):
9              adj[(x,y)] = 0
10     for EI in UGraph.Edges():
11         adj[(EI.GetSrcNId(),EI.GetDstNId())] = 1
12         adj[(EI.GetDstNId(), EI.GetSrcNId())] = 1

```

Figure 2: Code

```

13         for i in xrange(n):
14             adj[(i,i)] = 1
15         return adj
16
17 N=input("No. of vertices (n<4039):")
18 edges=input("No. of edges:")
19 adj = randomGraph(N,edges)
20 edges = sum([sum(x) for x in adj])/2
21 L = N/10
22
23 # declare your variables
24 cen = dict(zip([i for i in range(N)], [0 for i in range(N)]))
25 cen = LpVariable.dicts("cen",cen, 0, 1,LpInteger)
26
27 assignments = {}
28 for i in range(N):
29     for j in range(N):
30         assignments[(i,j)] = adj[(i,j)]
31
32 assignments = LpVariable.dicts("assignments",assignments,0,1)
33
34 # defines the problem
35 prob = LpProblem("problem", LpMinimize)
36
37 # defines the objective function to minimize : sum(y)
38 prob += lpSum([cen[i] for i in range(N)])
39
40 ##### Constraint definitions #####
41
42 # sum(x_ij) <= L*y[i]
43 for i in xrange(N):
44     prob += lpSum([assignments[(i,j)] for j in xrange(N)]) <= L*cen[i]
45
46 # sum(x_ij) = 1 for all j
47 for j in xrange(N):
48     prob += lpSum([assignments[(i,j)] for i in xrange(N)]) == 1
49
50 # y_i >= x_ij for all i,j
51 for i in xrange(N):
52     for j in xrange(N):
53         prob += assignments[(i,j)] <= cen[i]
54 #the assignment can be done only when the edge is present
55 for i in xrange(N):
56     for j in xrange(N):
57         prob += assignments[(i,j)] <= adj[(i,j)]
58
59 # solve the problem using GLPK
60 status = prob.solve(GLPK(msg=0))
61 print lpStatus[prob.status]
62 list_var=prob.variables()
63 list_cen=list_var[(len(list_var)-N):]
64 count=0
65 for v in list_cen:
66     if(v.varValue==1.0):
67         count+= 1
68 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

5 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 doesn't 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:

```

1:  $S \leftarrow v_1, v_2, \dots, v_k$ 
2:  $S' \leftarrow \text{one - swap vertex sets of } S$ 
3: while  $\text{MaxFlow}(G_{V_{\max}}, c_{V_{\max}}) > \text{MaxFlow}(G_S, c_S)$  do
4:    $S \leftarrow V_{\max}$ 
5:    $S' \leftarrow \text{one - swap vertex set of } S$ 
6:    $V_{\max} \leftarrow \text{argmax}_{V \in S} \text{MaxFlow}(G_V, c_V)$ 
7: end while
8: if  $\text{MaxFlow}(G_S, c_S) = n - k$  then
9:   for all  $v \in V$  do
10:    if  $f_{\max}(v, s) = 1$  then
11:       $h(v) = s$ 
12:    end if
13:  end for
14:  return  $S, h$ 
15: end if

```

The code for the above algorithm is as follows:

```

1  import networkx as nx
2  import snap
3  from random import shuffle
4  from collections import defaultdict
5
6  #Generating a random graph using snap
7  def randomGraph(n,edges):
8      UGraph = snap.GenRndGnm(snap.PUNGraph, n, edges)
9      adj = dict()
10     for x in range(n):
11         for y in range(n):
12             adj[(x,y)] = 0
13     for EI in UGraph.Edges():
14         adj[(EI.GetSrcId(),EI.GetDstId())] = 1
15         adj[(EI.GetDstId(), EI.GetSrcId())] = 1
16     for i in xrange(n):
17         adj[(i,i)] = 1
18     for x in range(n):
19         for y in range(n):
20             print adj[(x,y)],
21         print
22     return adj
23
24 #Storing the random graph in G
25 def getGraph(N,adj):
26     G = nx.Graph()
27     G.add_nodes_from([x for x in xrange(N)])
28     for i in xrange(N):

```

Figure 4: Code


```

29         for j in xrange(i,N):
30             if adj[(i,j)]:
31                 G.add_edge(i,j)
32     return G
33
34 #Obtaining a random set S of initial k centers
35 def getS(G,k):
36     nodes = G.nodes()
37     shuffle(nodes)
38     return nodes[:k]
39
40 #making FG a directed flowgraph to give as input to max flow functions ( s-->V-->S-->t )
41 def getFlowGraph(G, S, L):
42     FG = nx.DiGraph()
43     N = len(G.nodes())
44     #for common nodes in V and S make separate keys for S
45     for x in G.nodes():
46         FG.add_node(x)
47         if x in S:
48             FG.add_node(x + N)
49
50     FG.add_nodes_from(['s','t'])
51     edges = set(G.edges())
52     for s in S:
53         for v in G.nodes():
54             if (min(s,v),max(s,v)) in edges:
55                 FG.add_edge(v, s + N, capacity=1)
56
57     for v in G.nodes():
58         FG.add_edge('s',v, capacity=1)
59     for s in S:
60         FG.add_edge(s + N, 't', capacity=L)
61     return FG
62
63 #Obtaining the set Vmax with one swap from S to get the max flow value set
64 def getVmax(G, S, L):
65     V = G.nodes()
66     # flow[i][j] = max-flow in (S, V) when S[i] is swapped with V[j]
67     flow = [[None for _ in xrange(len(V))] for _ in xrange(len(S))]
68     setS = set(S)
69     for i in xrange(len(S)):
70         for j in xrange(len(V)):
71             if V[j] not in setS:
72                 temp = S[i]
73                 S[i] = V[j]
74                 flow_graph = getFlowGraph(G, S, L)
75                 flow[i][j] = nx.maximum_flow_value(flow_graph, 's', 't')
76                 S[i] = temp
77
78     max_value = 0
79     swap_pair = None
80     for i in xrange(len(S)):
81         for j in xrange(len(V)):
82             if max_value < flow[i][j]:
83                 swap_pair = (i,j)
84                 max_value = flow[i][j]
85
86     i,j = swap_pair
87     S[i] = V[j]
88     return S

```

Figure 9: Code

```

87
88 #changing S and Vmax by one swaps so as to achieve the max flow possible after each obtained S
89 def doOneSwaps(G, S, L):
90     Vmax = getVmax(G, S, L)
91     while nx.maximum_flow_value(getFlowGraph(G, Vmax, L), 's', 't') > nx.maximum_flow_value(getFlowGraph(G, S, L), 's', 't'):
92         S = Vmax
93         Vmax = getVmax(G, S, L)
94     return S
95
96 N, M = 10,30
97 adj = randomGraph(N,M)
98 k, L = 5,3
99 G = getGraph(N,adj)
100 S = getS(G,k)
101 S = doOneSwaps(G, S, L)
102 H = defaultdict(list)
103 #max flow value cannot be greater than the number of nodes in the graph or set V
104 if nx.maximum_flow_value(getFlowGraph(G, S, L), 's', 't') == N:
105     _,flows = nx.maximum_flow(getFlowGraph(G, S, L), 's', 't')
106     edges = set(G.edges())
107     #if a flow exists along a particular path assigning v to that vertex in S
108     for v in G.nodes():
109         for s in S:
110             if (min(v,s),max(v,s)) in edges:
111                 unitflow = False
112                 try:
113                     if flows[max(v,s + N)][min(v,s + N)] == 1:
114                         unitflow = True
115                 except KeyError:
116                     try:
117                         if flows[min(v,s + N)][max(v,s + N)] == 1:
118                             unitflow = True
119                     except KeyError:
120                         pass
121                 if unitflow:
122                     H[s].append(v)
123 else:
124     print "Failed!"
125 print S
126 print H

```

Figure 6: Code

6 Graph Generation

6.1 Random Graph

For testing the implemented algorithms on random graphs, SNAP, a Python based system developed by Stanford University was used which takes the number of nodes and edges as input and gives a random subgraph of a data set and represent it as an

adjacency matrix, ready to be fed into the LP and local search algorithm.

```
import snap
#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
    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
```

Figure 7:

6.2 Star Graph

In an attempt to try our programs on a more clustered data set, this program was implemented to create sparsely-connected star graphs, with edges chosen to be added with a probability p .

```

1 import snap
2 import random
3 import networkx as nx
4 from collections import namedtuple
5
6 class Graph(object):
7     def __init__(self, specs):
8         self.k = specs.k
9         self.l = specs.l
10
11         if specs.struct == 'stars':
12             self.N = self.k*self.l
13             self.p = specs.p# supplementary edge probability
14             self.starGraph()
15
16         elif specs.struct == 'random':
17             self.N = specs.N
18             self.edges = specs.M
19             self.randomGraph()
20
21         else:
22             raise ValueError('Invalid specifications!')
23
24         self.nxGraph = self.NXify()
25
26     def initAdj(self):
27         self.adj = dict()
28         for x in range(self.N):
29             for y in range(self.N):
30                 self.adj[(x,y)] = 0
31         for i in xrange(self.N):
32             self.adj[(i,i)] = 1
33

```

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
52         for center, clients in zip(star_centers, star_clients):
53             for client in clients:
54                 self.adj[(center,client)] = self.adj[(client, center)] = 1
55
56         for x in range(self.N):
57             for y in range(x + 1,self.N):
58                 if self.adj[(x,y)] != 1:
59                     self.adj[(x,y)] = int(random.random() < self.p)
60                     self.adj[(y,x)] = int(random.random() < self.p)
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
71     return G

```

Figure 9: Code

6.3 Unit Square

7 Observations made

This section is a collection of all our trials and errors during our exploration of this topic along with few of the concepts learnt which provide the base for these trials.

7.1 Lower Bound on K

Lets say we make disjoint set of partitions of our input graph nodes such that each partition has two types of vertices. Inner and boundary, the inner vertices are the ones which have no edges outside its partition, while the boundary vertices are vertices which do have edges to other partitions. Let the i^{th} partition be represented by its inner vertices C_i and its boundary vertices as $B(C_i)$.

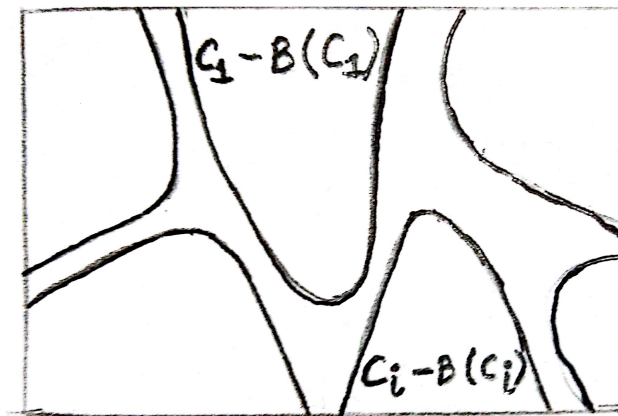
Since we know K vertices can cover up to L vertices so the max vertices than can be covered is $V = K * L$. Hence the minimum K required to cover V vertices is $\lceil V/L \rceil$. This is a naive lower bound on K . But using this concept, we can get a better lower bound.

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.

The above bound has been presented in a research paper already. We observed that, the above mentioned paper is not involving the boundary vertices in formulating a lower bound. So, we toyed for some time with ideas on involving the boundary vertices. We thought using the residual 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 vertices. But all of this depends upon the connectivity of the particular graph and what kind of partition we take which did not look promising after a point of time either.

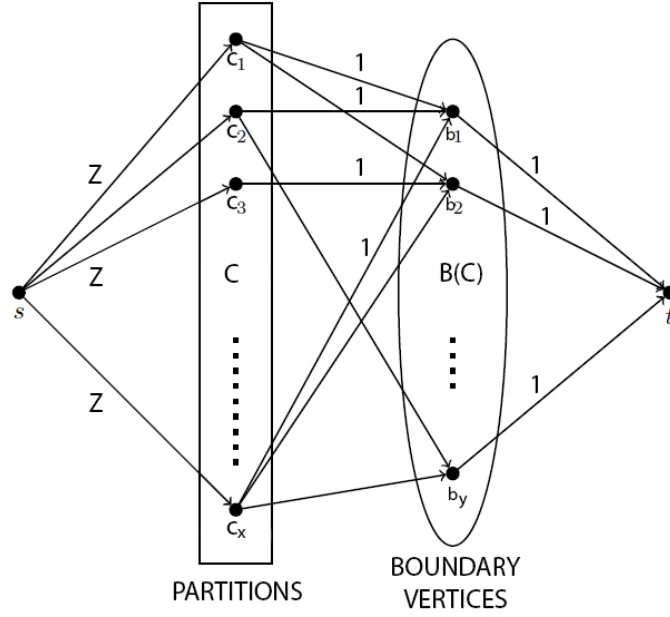
Later when Jatin and Aounon started thinking towards this way we again applied our thinking on the same.

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, $\lceil (C_i - B(C_i))/L \rceil * L - (C_i - B(C_i))$, can be used to cover the boundary vertices.

To do this, a flow network is created:



Here Z refers to the remaining capacity of the centers as put in the inner vertices i.e $\lceil (C_i - B(C_i)) / L \rceil * L - (C_i - B(C_i))$. Now there are two cases:

- If we are not able to find a flow to cover all the boundary vertices within a distance of 2 with the residual capacities, then there exist a set of partitions P_1, P_2, \dots, P_i and $B(P_1), B(P_2), \dots, B(P_i)$ for which Hall's theorem does not hold i.e.:

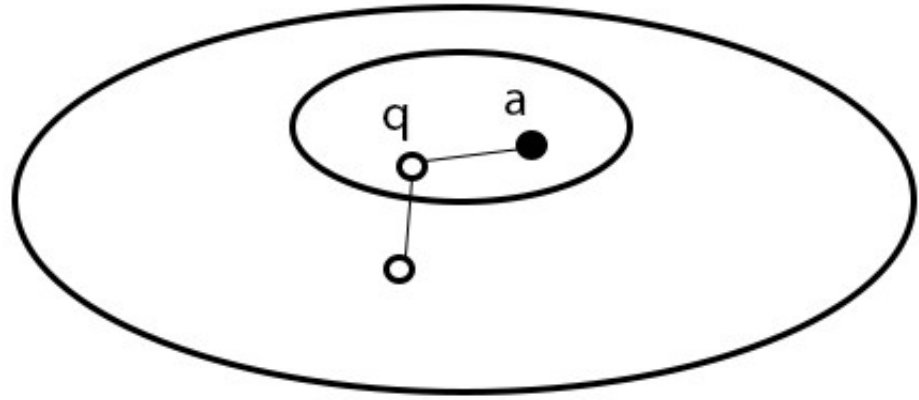
$$|B| > \sum_{i=1}^N \lceil (P_i - B(P_i)) / L \rceil * L - (P_i - B(P_i))$$

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 will always be in the interior of the new set of partitions, because if they were not then it can be covered by a residual capacity, a contradiction.

- 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 all the necessary residual capacity which is actually taken into account by the flow network. Our attempt to resolve this:

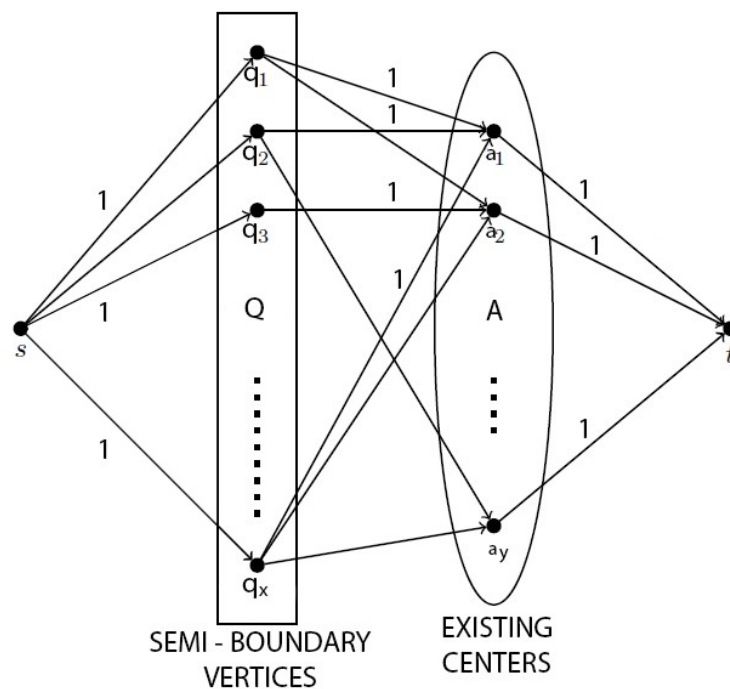


We define

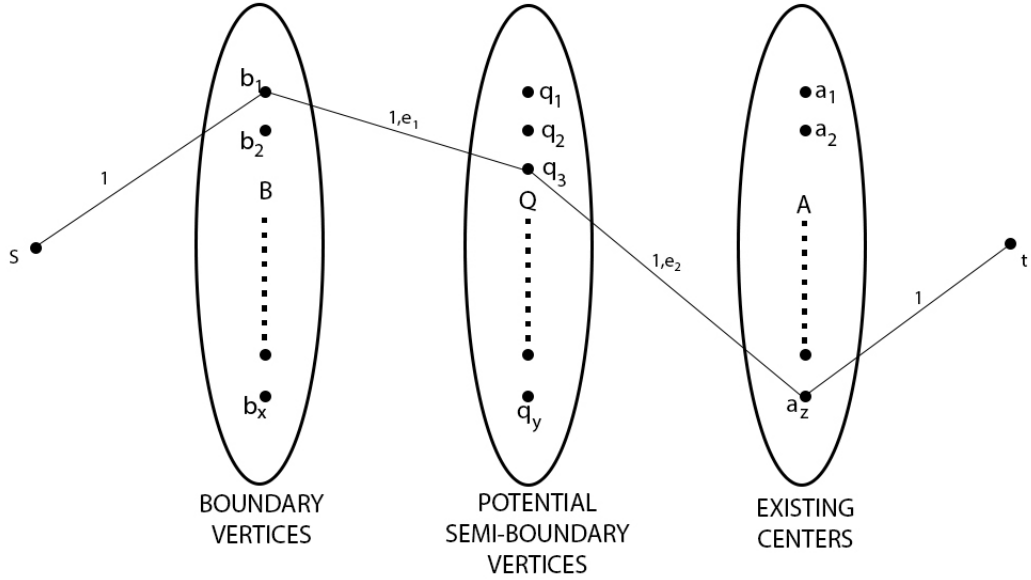
- Node type: a : A center which does not have edge to any boundary vertex.
- Node type: q : An interior non - center, which is a client of a and adjacent to a boundary vertex. We will call it the semi boundary vertex.

We observe that if we make as many semi-boundary centers as possible, such that they cover maximum number of boundary vertices, by shifting the centers located at a to q 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 its respective center a , but now the minimum maximum distance would be at most $4 \cdot \text{OPT}$. Also, we would need to shift to such a set of semi-boundary vertices that maximises the number of distinct boundary vertices getting covered.

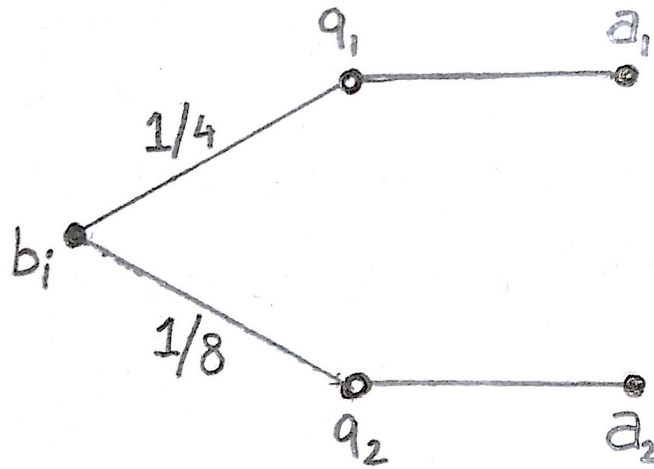
So, we make a flow network:



But we cannot make greedy moves and just move to the semi-boundary vertices with maximum boundary neighbors, as it is very necessary to make sure these boundary neighbours are distinct. To ensure this, we modify the above flow network:



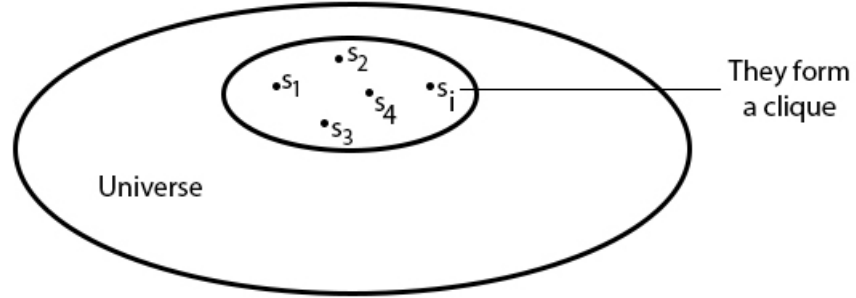
We keep the capacity of all edges in this flow network as 1 but modify the edge weights of edge weights between B,Q and A to keep a check. Between A and Q, e_1 the remaining capacity of q_i which is $-L$ —the number of non centers transferred to it through a_i is taken as the edge weight between q_i and a_i . While the edge weight between B and Q, e_2 is equal to the inverse of the number of boundary vertices that the corresponding q will cover if selected.



In a case as shown above in the figure if capacity permits, we will choose $e_2 = 1/8$ as it is covering 7 other boundary vertices except the one in the selected augmented path. Also we will have to make changes to the edge weights like

converting $1/4$ to zero and the corresponding edges of the same semi boundary vertex involved to $1/3$ and the semi-boundary to corresponding center as 3. If later $1/8$ weight wedge is not a part of the max flow then we simply revert back to the original weights.

But we realised that this problem of covering maximum number of boundary vertices is a problem reducible to the max-coverage problem, which is NP-hard. Hence the problem we are trying to solve is indeed 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 which is not possible.

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

7.2 Independent Trial

- 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 & Schmoys. 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.

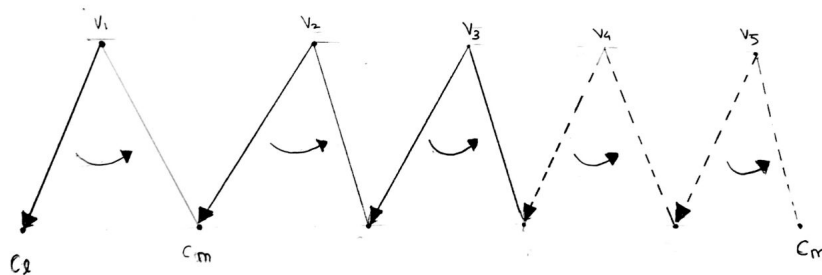


Figure 10: reassignment through propagation

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 ($m \neq l$) 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 areas 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 $j=2$ whereas the situation is the other extreme for the other centers.

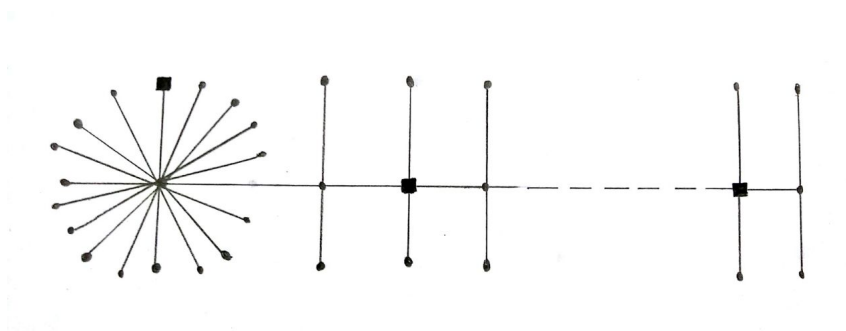


Figure 11: Counter Example

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.

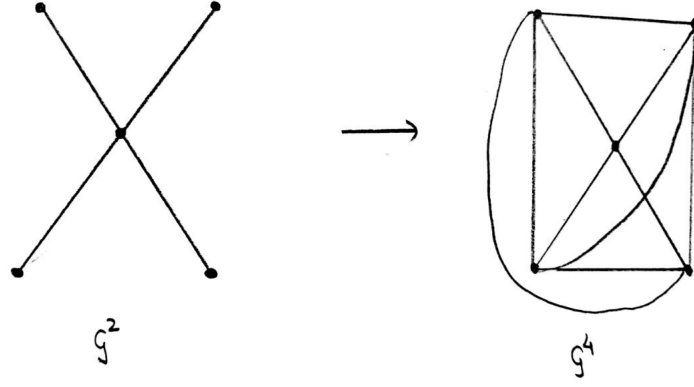


Figure 12: Clique formation

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, centers in G^2 are represented by square labels in the left figure and diamond label on the right while the one that replaces them in G^4 is represented by a square in the right figure.

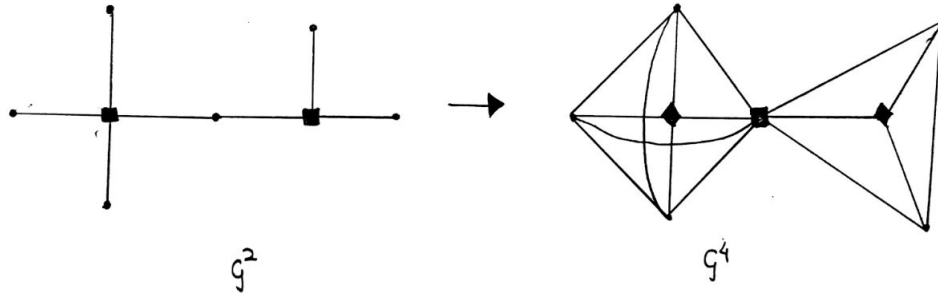
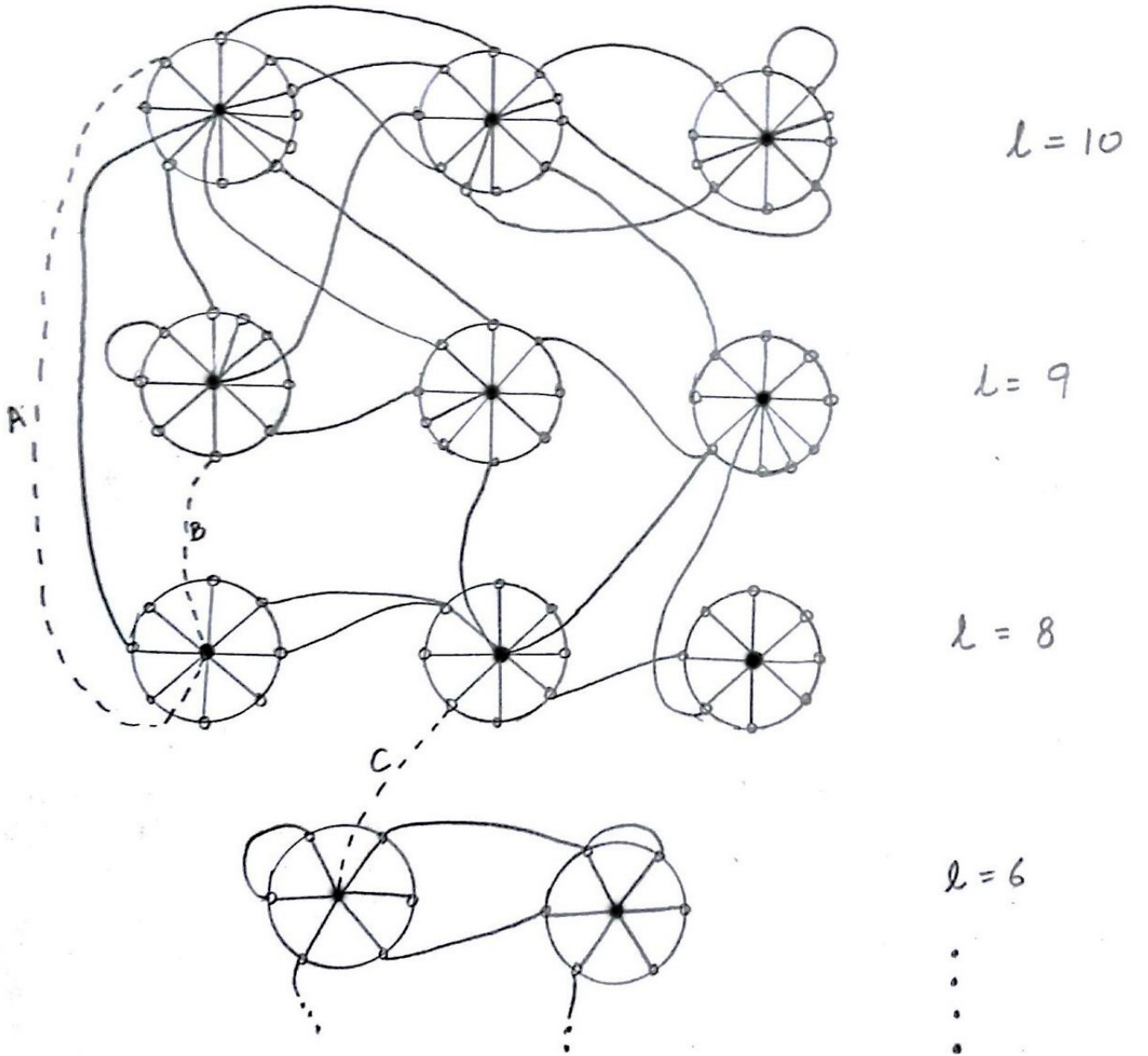


Figure 13: Merging

Why are we doing this? By doing the above *merging* operation, we obtain one extra center that can be placed at any location of our choice. 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 tried to come up with some kind of data structure that might help us make some claims on the lower bound on K , when the edge reassignment operation stops at a local optima.



As shown above, we make a level structure with the set domains with center C_i by placing the domains of size l at level l . This is the final structure after all reassignments which follows the upcoming theorems. Also edges A, B and C cannot be present.

We say it is the final structure because, after a reassignment operation, the set of neighbors of (more than or equal to 2) centers change, and also, their capacities change, thus moving them to different levels throughout the edge reassignment phase.

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 $l' \geq l + 2$. 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.

Theorem: When there can be no more reassignments, there would not be any edge from a level l domain to other level below it if its non-center has an edge to the center of $l+1$.

Proof: Taking the theorem to be false, if such was our situation then it could end up bringing all 3 at the same level by few reassignments thereby decreasing the highest degree centers. But we have already said no more reassignments are possible. Hence, a contradiction.