Capacitated K-Centre Problem

SUMMER INTERNSHIP REPORT

May 9th - July 11th, 2016

INDIAN INSTITUTE OF TECHNOLOGY, DELHI

Authors:
Anya Chaturvedi
K.R. Prajwal
Sagar Sahni
Ketaki Vaidya

Supervisor:
Dr. Naveen Garg
Professor
Dept. of CSE
IIT Delhi

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY DELHI

Certificate

This is to certify that the students whose names are given below have done their summer internship on **The Capacitated K-Centre Problem** under my guidance for a duration of two months in Summer - 2016 and have submitted this report.

Name	Institute
Anya Chaturvedi	Motilal Nehru National Institute of Technology, Allahabad
K.R. Prajwal	National Institute of Technology, Tiruchirappalli
Sagar Sahni	SRM University, Kattankulathur
Ketaki Vaidya	National Institute of Technology, Silchar

Date:

Dr. Naveen Garg
(Supervisor)

ACKNOWLEDGEMENTS

We wish to express our sincere gratitude to Professor Naveen Garg for letting us be a part of his research group for the summer.

We also wish to thank the research scholars, Jatin, Aounon and Nikhil for their continuous help and guidance in carrying out our implementations and giving us topics to brainstorm upon.

Thank you to all the above mentioned for taking out time for us and providing us with an enriching and memorable experience.

- Anya, Prajwal, Sagar & Ketaki.

Contents

1 Abstract

The capacitated K-centre problem is basically a facility location problem, where one is asked to assign K points to K facilities in a given distance metric. In doing this we minimise 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, counterexamples that we studied and a few basic implementations of the same. Starting with a more formal description of our problem, we describe the flow method which helps us verify a solution to the problem. Next, we show how we implemented the integer program and a partially correct local search algorithm. In implementing the above programs we have used

three ways for 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 a few observations which may not be very conclusive but are our attempts to solve the problem over the summer.

2 A Look at the Problem

The problem studied by us is actually a generalization of the K-centre problem which we explain here.

2.1 The K-Centre Problem

Given n vertices with specified distances, one wants to build K facilities at different vertices such that each vertex has access to a facility and minimise the maximum distance between each vertex and the corresponding 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-centre, methods have been presented for obtaining an approximation factor of 2. Given a complete undirected graph $G = (V, E)$ with distances $d(v_i, v_j) \in \mathbb{N}$ satisfying the triangle inequality, we have to find a subset $S \subseteq V$ such that $|S| = k$ and such that it covers all V.

Input:
1. A metric space, or, in other words, a complete graph that satisfies the triangle inequality.
2. An upper bound on the number of centres, K.

Output:

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

where d is the distance function.

This is quite similar to the problem of placing K disks such that all points are covered in the set V and the maximum radius required to do so is minimised.

2.2 The Capacitated K-Centre

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

Input:
1. A metric space, or, in other words, a complete graph that satisfies the triangle inequality.
2. An upper bound on the number of centres, K.

3. A maximum capacity for each centre, L.

Output:

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

such that,

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

where,

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

The first constant factor 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 centres at any single vertex without counting the center itself while calculating the capacity used.

Initially we went through the sections 2.6 and 9.3 of Williamson & Shmoys which solve the problem of finding the minimum degree spanning tree problem which is also NP-hard. Here we learnt of methods like local search used to create approximation algorithms. Next we read a few papers related to our problem and gathered what is presented in the upcoming sections.

3 Checking the Feasibility of the Input Graph

We say that if we solve the capacitated K-centre 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 $c : V \rightarrow N$ we say that $S \subseteq V$ is a capacitated dominating set if there exists a mapping $f : V \rightarrow S$ (domination mapping) such that:

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

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

In the new version of the K-centre problem, the goal is to get a set S of K vertices and an assignment $h : V \rightarrow S$ of every vertex (in V) to an open centre such that the greatest distance between a vertex and the centre it is assigned to is minimised and no facility is assigned more vertices than its capacity. Note that the assignment is not for all vertices in V. We make the assumption that every centre 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-centre 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-centre 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-centre problem.

However, note that the capacitated dominating set problem is an NP-complete problem. Therefore, 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 $c : V \to N$, checking whether $S \in V$ is a capacitated dominating set can be viewed as a network problem. We construct a directed graph $G_S = (V_S, E_S)$ as follows:

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

Let $c_S : E_S \to N\mathbb{N}$ be the capacity function such that,

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

As defined above, 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, from each vertex in V 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.
The flow network as described has been shown in Figure 1.

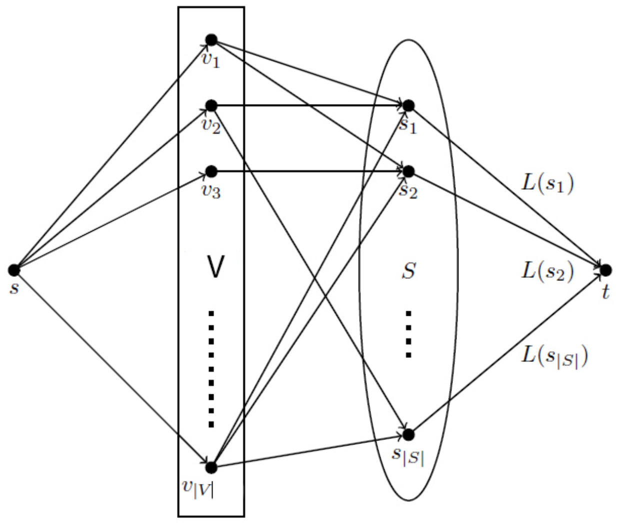


Figure 1: The flow network

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

4 The Integer Program

Linear programming is a mathematical technique for maximising or minimising 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 integral. 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 makes it a subset of linear programming. Ours is a problem where we wish to minimise the maximum distance between a vertex u and its assigned centre thus our integer program is framed as:

Objective Function:

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

Variables:

1. ***cen*** - an integer dictionary which specifies the centres among the vertices where if $cen[i] = 1$ then the i^{th} index vertex is taken to be a centre otherwise not.

2. ***assignments*** - a dictionary, $assignments[i][j] = 1$ will represent that the j^th index vertex is assigned to the centre created at the i^th index vertex, we initialise it with the adjacency matrix

3. N - 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 centre

2. $\forall j \sum_{i=1}^{N} assignments[i][j] = 1$
   ...a vertex can be assigned only to a single centre

3. $\forall i \, \forall j \, assignments[i][j] \leq cen[i]$
   ...a vertex can only be assigned to a vertex if the later is a centre

4. $\forall i, \forall j \, assignments[i][j] \leq adj[i][j]$
   ...assignment of vertices can only be done if the edge between a vertex and its corresponding centre is present in the original graph

You can refer to the code in Appendix A.

5 Local Search Algorithm

Local search can be used on problems that can be formulated as finding a solution by maximising 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 which is actually a randomised algorithm. It does not work on a set of graphs but by increasing the number of iterations on the same graph it gradually gives the solution for most as the probability involved increases.

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 maximises the number of dominated vertices. If one of the set dominates more vertices than current 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: $S \leftarrow v1, v2, \ldots., vk$
2: $S' \leftarrow one - swap\ vertex\ sets\ of\ S$
3: while $MaxFlow(G_{V_{max}}, c_{V_{max}}) > MaxFlow(G_S, c_S)$ do
4:    $S \leftarrow V_{max}$
5:    $S' \leftarrow one - swap\ vertex\ sets\ of\ S$
6:    $V_{max} \leftarrow argmax_{V \in S} MaxFlow(G_V, c_V)$
7: end while
8: if $MaxFlow(G_S, c_S) = n - k$ 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
```

You can refer to the code in Appendix B.

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. It takes the number of nodes and edges as input and gives a random sub graph of a data set which we convert represents to an adjacency matrix, ready to be fed into the Integer Program and local search algorithm.

You can refer to the code in Appendix C.

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.

We were able to get 100% accuracy and this was the strongest dataset we tested upon.

You can refer to the code in Appendix D.

6.3 Unit Square

On discussing our above mentioned methods with Sir he suggested another way for making the input graphs which he felt might give us better results for further analysis. The method proceeds by taking N random points in a unit square. The edge weight between all possible pairs of these N points are calculated by using different L^p norms, for $p \geq 1$. W is chosen which denotes the maximum edge weight allowed in a particular thus reducing the number of edges in the otherwise obtained complete graph. The graph is built with the N random points as nodes and edges (a, b) if distance between (a, b) is $\leq W$, for all $a, b \in N$. This graph is used to obtain the optimal K using a capacity, which we have taken as $L = N/10$.

We have used the same graph to check if the randomised local search algorithm yields the optimal solution with the above obtained K and L.

Even though this did not strike at first on why this will be in any manner be different than using random graphs, still we decided to proceed with the implementation. During our visualization of the graphs created, we saw that this method gave us a very large variety of graphs, in the sense that the graphs had local clusters, articulation points, isolated vertices, cliques and a wide variation in the degrees of the nodes.

We checked it on the integer program, local search and randomised local search programs. Mostly it worked but took a long time as we have iterate the local search for 5-10 iterations to increase the chance of it being successful. If somehow the complexity can be reduced we can see how well it works for larger graphs. The complexity as of now is the order of O(n^7). At present it is working very well for below 40 nodes giving almost a 100% accuracy but after we clocked it up to 80 nodes(we could not increase more) we found that the accuracy drops sharply to 98-99% i.e. one or two failures for every 110 nodes. Though it gives failure few times it is difficult to analyse due to the large number of vertices in the failed examples.

Further, we came up with minor hacks and tweaks to get the best of randomization and non-randomization in our algorithm by including them in a single run and could get all correct on several test sets of 110 unit square graphs each. However, this still exists as a minor tweak, and we are not able to analyse pictorially for larger graphs such as 80 nodes as we did for $\leq$ 40 nodes. We hence, tried a different approach of representing it as a bipartite graph, but that was not possible through the networkx library which we have used. We hence would need a better method to analyse large graphs of 80 nodes or more.

You can refer to the code in Appendix E.

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

Let us 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 with vertices outside their partition, while the boundary vertices are the ones 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 maximum vertices that 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 may 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 centres in the partition itself for such vertices. If we say there are p partitions then $\sum_{i=1}^{p} \lceil (C_i - B(C_i))/L \rceil$ vertices are required at least 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 centres or shifting the centres 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.

Let us say we partition the graph into sets of nodes $C_i - B(C_i)$ $1 \leq i \leq N$ with the boundary vertices outside the partitions as shown in Figure 2.

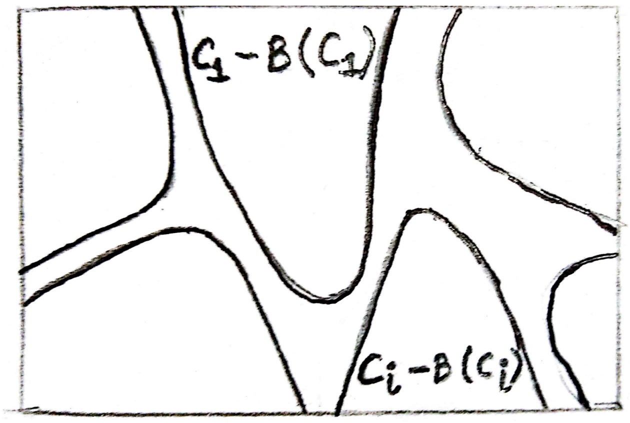


Figure 2: Partitioning the Vertex Set

As discussed before, the minimum number of centres 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, $\lceil (C_i - B(C_i))/L \rceil * L - (C_i - B(C_i))$ and can be used to cover the boundary vertices. To implement this, a flow network is created as labelled in Figure 3.

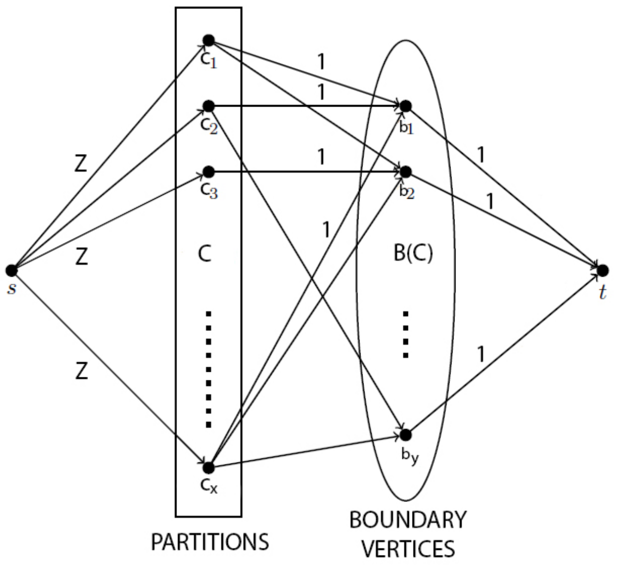


Figure 3: $Z = \lceil (C_i - B(C_i))/L \rceil * L - (C_i - B(C_i))$

Here Z refers to the remaining capacity of the centres as opened 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 centres 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:

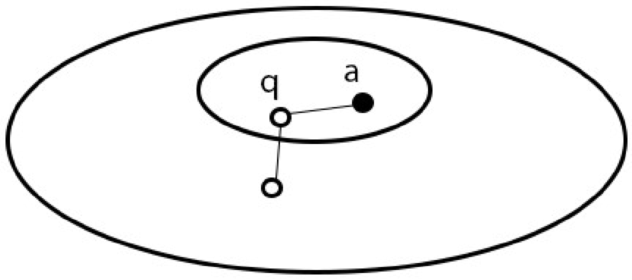


Figure 4: Reassignment of Centres

We define (the nodes labelled in Figure 4) as:

- Node type: a : A centre in a partition which does not have edge to any boundary vertex.
- Node type: q : An interior non - centre, 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 centres as possible, such that they cover maximum number of boundary vertices, by shifting the centres located at a to q then we can come to a conclusion. If we make the semi-boundary vertex q as a centre, then it will cover all the clients that have been covered by its respective centre a, but now the minimum maximum distance would be at most 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 for the same as shown in Figure 5.

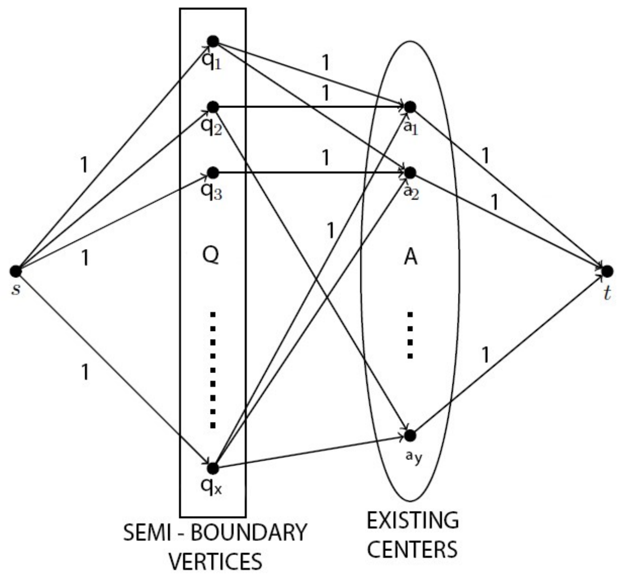


Figure 5: Flow to exchange Semi-boundary vertices with Existing Centres

But we cannot make greedy moves and just move to the semi-boundary vertices with maximum boundary neighbours, as it is very necessary to make sure these boundary neighbours are distinct. To ensure this, we have Figure 6.

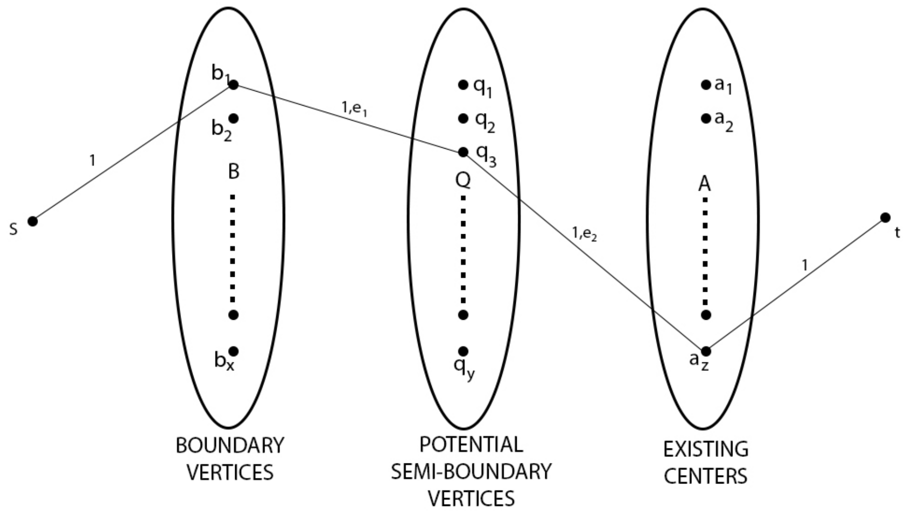


Figure 6: New Flow Method

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 centres 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.

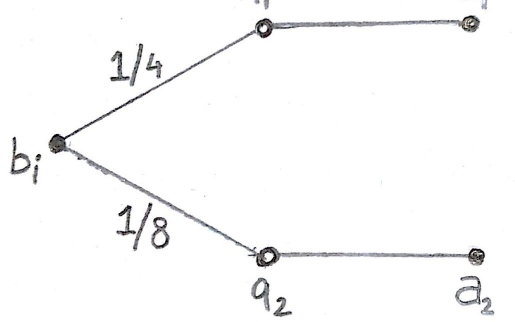


Figure 7: Case of 2 Available Paths

In a case as shown above in Figure 7, if capacity permits, we will choose the path with $e_2 = 1/8$ as it the semi-boundary vertex involved 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 centre 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. (Figure 8)

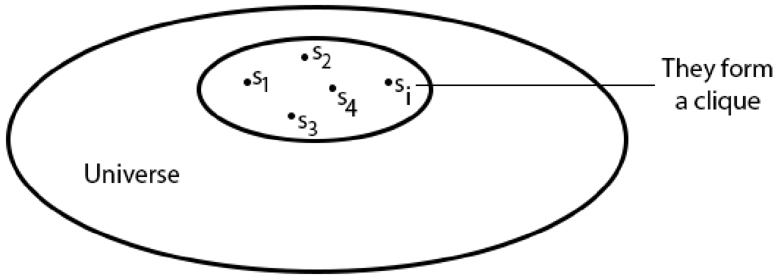


Figure 8: Max-Coverage Problem

If there is a way where we can get minimum subsets to cover all of the universe then it is the same solution as we were trying to form which is not possible. However, we still feel that this idea of shifting centres to semi-boundary vertices could lead to some result.

7.2 Independent Trial

In our attempt we start by applying the 2-approximation *K*-centre algorithm on the given graph. We then try 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 & Shmoys. Effectively,if possible we transfer 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.

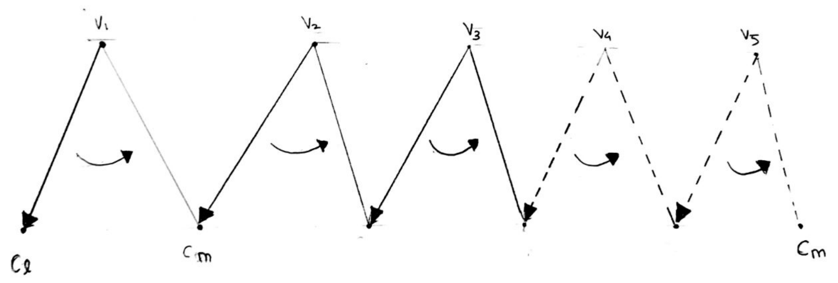


Figure 9: Reassignment Through Propagation

Algorithm:

```
1:  Start with $K'$ centres of $G^2$ with no regard to capacity $L$ using the 2-approximation
    algorithm for uncapacitated $K$-centre.
2:  $C \leftarrow$ set of all $K$-centres
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 centre
6:          $C_l \leftarrow$ set of all centres with degree $l$
7:  Subphase
8:  **while** there are centre with degree $l$ **do**
9:      Choose a client v assigned to some centre in $C_l$.
10:     Choose any other centre $C_m$, preferably, of maximum residual capacity, and find
        a path as described in Figure 9 to $C_m$. Choose the path to $C_m$ having centre-client
        pairs such that we can do a series of edge swaps as described in Figure 9.
11: **end while**
```

This is done in order to reduce the maximum number of assignments to a particular centre so as to reach the capacity L. This propagation is possible only when a non centre is connected to two or more centres and hence reassignment can be done.
But we still might have too few centres in dense areas while just sufficient in other parts or maybe even having residual capacities in sparse areas. We also have a counterexample for the same as shown in Figure 10. The centres are represented by the rectangular marks on the vertices. The leftmost centre is covering many clients within a distance of ≤ 2 whereas the situation is the other extreme i.e. too few clients for the other centres. Thus wasting their capacity.

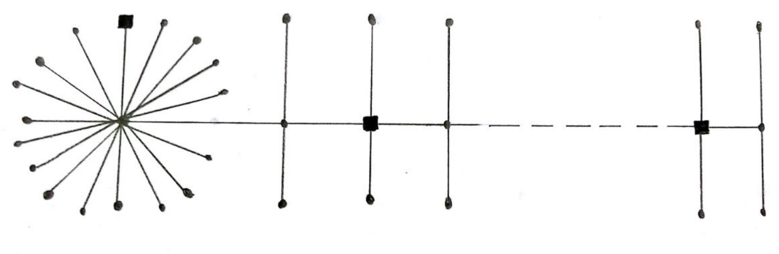


Figure 10: Counterexample

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

Hence, from G^2 we move to G^4. Here, each centre with its clients forming a star graph will transform into a clique. The same is shown in Figure 11 below.

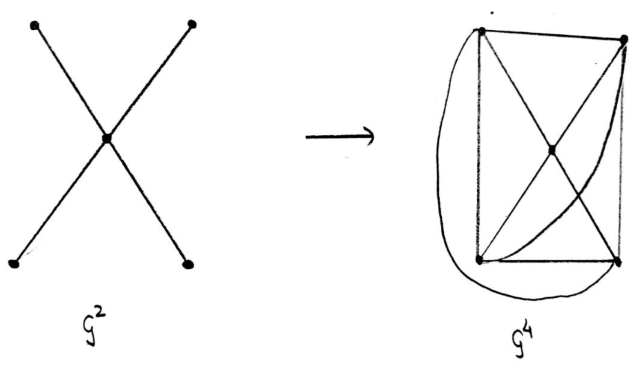


Figure 11: Clique formation

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

Following this we have free centres available which we can place in dense cliques with high degree of vertices. Like in Figure 12, centres 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.

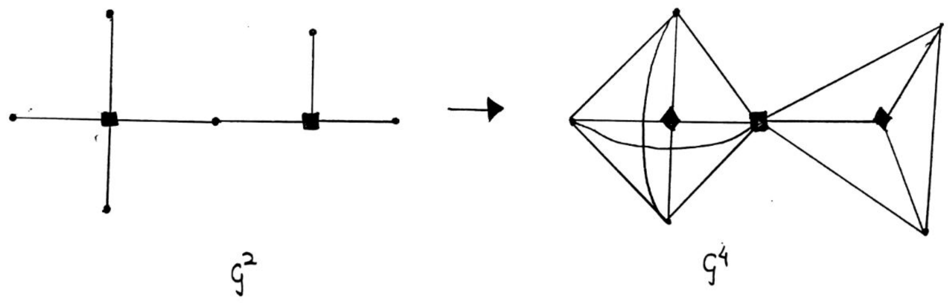


Figure 12: Merging

Why are we doing this? By doing the above *merging* operation, we obtain an extra centre that can be placed at any location of our choice. The problem still remains on how to keep obtaining such centres other than by merging and then placing them at an appropriate position. We stop here.

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.

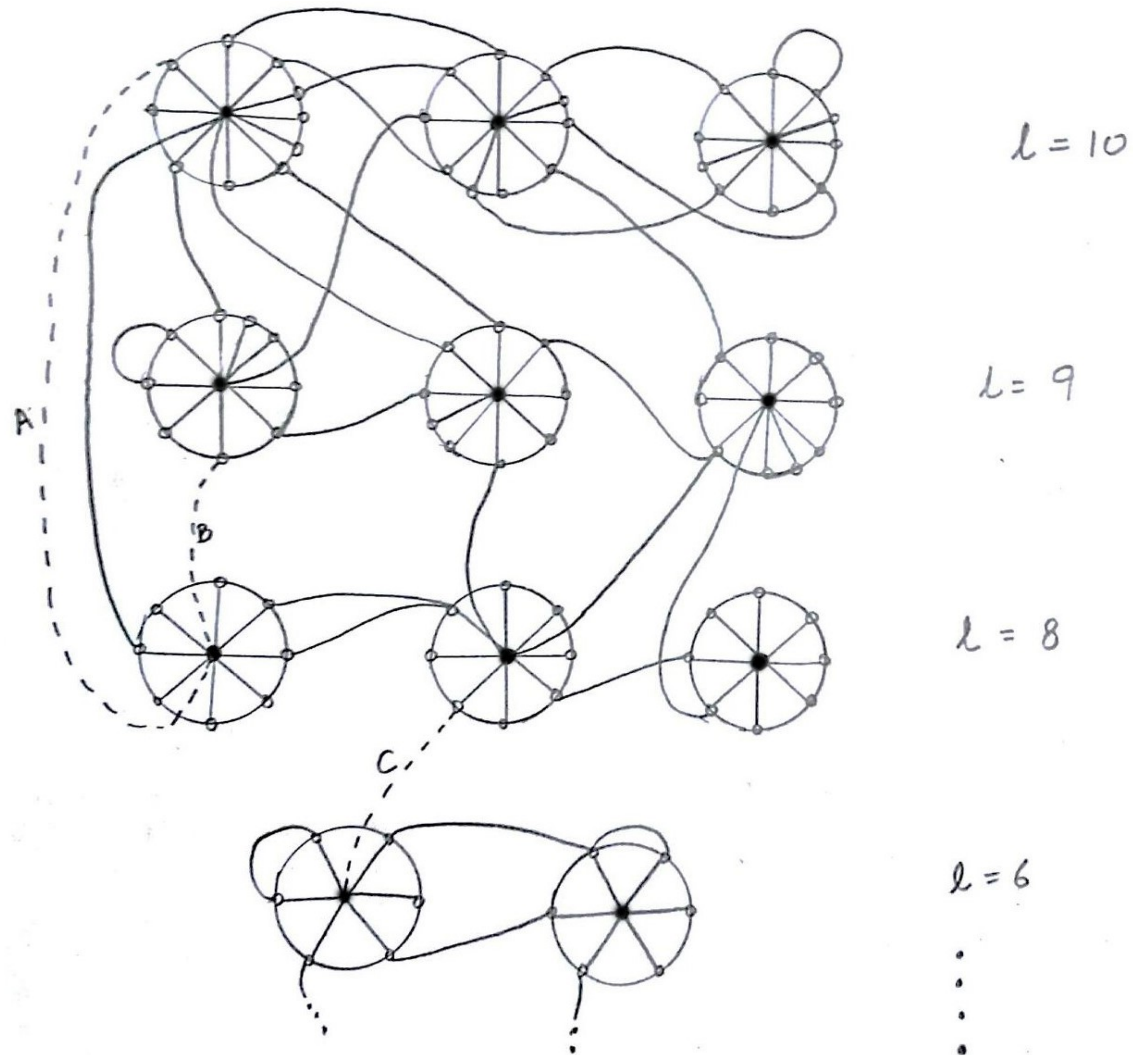


Figure 13: Level Structure

As shown above in Figure 13, we make a level structure with the set domains with centre 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 neighbours of (more than or equal to 2) centres 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 centre at level l is within a distance ≤ 4 to any node in level $l' \geq l+2$ for all l.

Proof: This case is depicted in the level structure figure by edge A in the level structure. After there can be no more reassignments, let there exists such a centre C at a level l which has an edge to some node P belonging to some level $l' \geq l+2$ i.e. let edge A exist. Let C' be the centre 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, if a centre C_l at level l is within distance ≤ 4 to a non-centre at level $l+1$, then any centre $C' < l$ at levels $< l$, will not be within a distance $\leq$4 from any node in the domain of C_l.

Proof: This situation is represented by edge B in the level structure. We will prove by contradiction. Let us say such a case exists, i.e., edge B in the figure exists. Then, we can do a series of reassignments, to reduce the degree of a centre at C_l at some level $\geq l+1$. This is a contradiction, since we assumed in the beginning that we have reached at least a local optima and there can be no more reassignments.

8 Bibliography

1. Samir Khuller, and Yoram J. Sussmann, *'The Capacitated K-Center Problem'*.

2. Hyung-Chan An, Aditya Bhaskara, and Ola Svensson, *'Centrality of Trees for Capacitated k-Center'*

3. David P. Williamson, and David B. Shmoys, *The Design of Approximation Algorithms*

4. Vijay V. Vazirani, *Approximation Algorithms*

5. Aounon Kumar, *Network Location Problem Notes* (Work in Progress)

Appendices

A The Integer Program

```
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
	'''for x in range(n):
		for y in range(n):
			print adj[(x,y)],
		print'''
	return adj

N, M = 10, 30
adj = randomGraph(N, M)
L = 3

# 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)

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]
'''for v in prob.variables():
    print(v.name, "=", v.varValue)'''
```

B Local Search Program

```python
import networkx as nx
import snap
from random import shuffle, randint
from collections import defaultdict
import matplotlib.pyplot as plt
import pprint, pickle
from ls_randomized import main as randomizedLS

#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

#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
```

```python
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():
                if x in S:
                        FG.add_node(x + N)
                FG.add_node(x)

        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)
        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, L), 's', 't'):
        S = Vmax
        Vmax = getVmax(G, S[:], L)
    return S

def main(N,adj,k,L):
    iterations = 5
    result = 'Failed'
    while iterations > 0 and result == 'Failed':
        print iterations
        G = getGraph(N,adj)
        S = getS(G,k)
        S = doOneSwaps(G, S, L)
        H = defaultdict(list)
        result = None
        #max flow value cannot be greater than the number of nodes in the graph or set V
```

```python
        if nx.maximum_flow_value(getFlowGraph(G, S, L), 's', 't') == N:
            # uncomment this to get assignments
            '''_,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)'''
            result = 'Success'
        else:
            result = 'Failed'
        iterations -= 1

    if result == 'Failed':
        ''' Do randomization if the max-cover approach fails'''
        result = randomizedLS(N, adj, k, L)

    if result == 'Failed':
        ''' Save the graph as an image, and its adjacency matrix
            as a pickle file '''
        pos = nx.spring_layout(G)
        nx.draw_networkx_nodes(G, pos, nodelist=S, node_color='b', node_size=200)
        nx.draw_networkx_nodes(G, pos, nodelist=list(set(G.nodes()) - set(S)), node_color='r', node_size=200)
        nx.draw_networkx_edges(G, pos, edgeList=list(G.edges()))
        fname = str(randint(1,10**9))
```

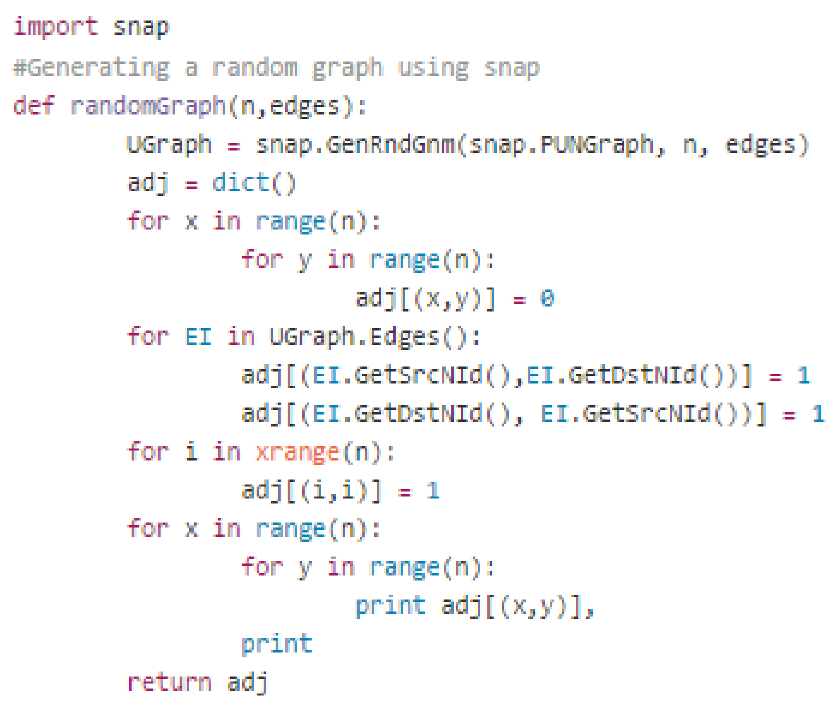
```
            plt.savefig('failures/' + fname)
            a = [[0 for _ in xrange(N)] for _ in xrange(N)]
            for key, val in adj.iteritems():
                a[key[0]][key[1]] = val
            pickle.dump(a,open('failures/' + fname,'w'))

    return result
```

For the randomized local search we modify the getVmax function in the above code as:

```
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)
    without_swap_flow = nx.maximum_flow_value(getFlowGraph(G, S, L), 's', 't')
    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') > without_swap_flow
                S[i] = temp

    swap_pair = []
    for i in xrange(len(S)):
        for j in xrange(len(V)):
            swap_pair.append([i,j])

    if swap_pair:
        i,j = choice(swap_pair)
    else: return S
    S[i] = V[j]
    return S

```

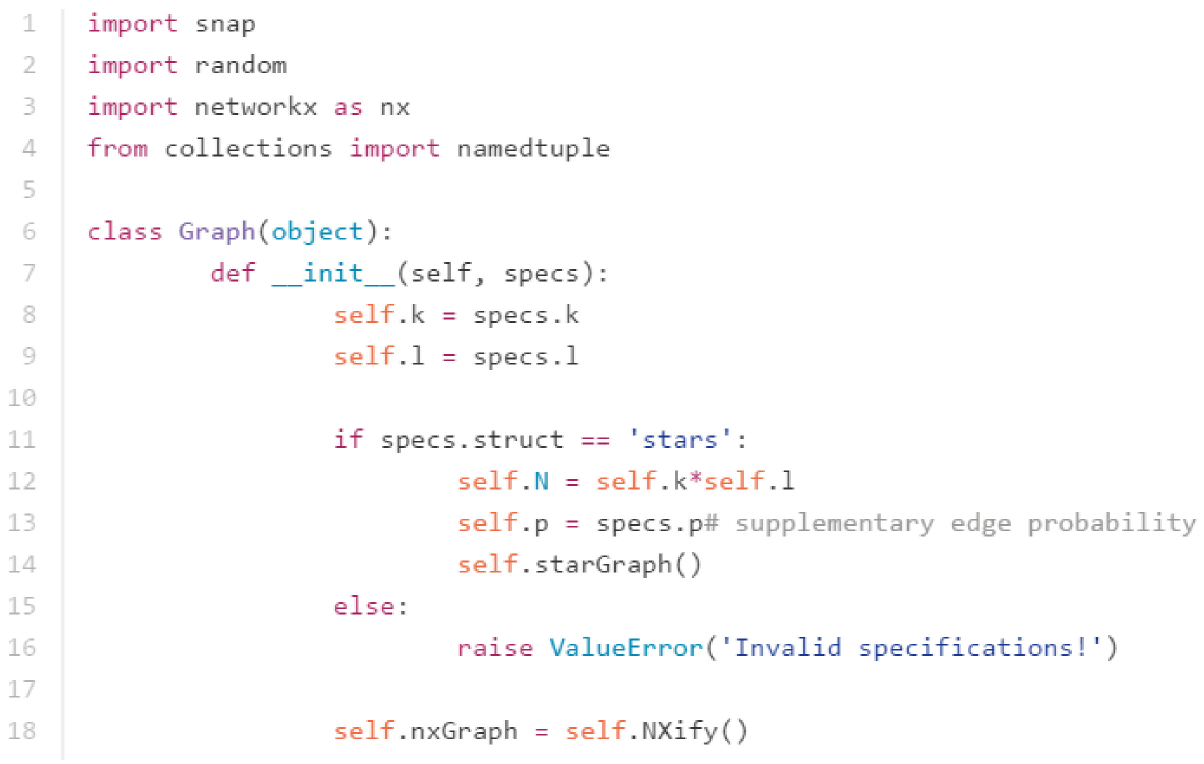
C Random Graph Generation

This is the code snippet from the program it was used in:



```
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
```

D Star Graph Generation



```
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()
        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

    def starGraph(self):
        self.initAdj()
        non_centers = set([x for x in xrange(self.N)])
        star_centers = random.sample(list(non_centers), self.k)
        for x in star_centers: non_centers.remove(x)
        non_centers = list(non_centers)
        random.shuffle(non_centers)
        star_clients = [non_centers[i:i + self.l - 1] for i in xrange(0, len(non_centers), self.l - 1)]

        for center, clients in zip(star_centers, star_clients):
            for client in clients:
                self.adj[(center,client)] = self.adj[(client, center)] = 1

        for x in range(self.N):
            for y in range(x + 1,self.N):
                if self.adj[(x,y)] != 1:
                    self.adj[(x,y)] = int(random.random() < self.p)
                    self.adj[(y,x)] = int(random.random() < self.p)


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

E Unit Square Graph Generation

```python
from random import random
import networkx as nx

class UnitSquareGraph(object):
	def __init__(self, N, P, W):
		self.N = N
		self.P = P
		self.W = W
		points = [[random(),random()] for _ in xrange(self.N)]
		self.dist = [[None for _ in xrange(self.N)] for _ in xrange(self.N)
		for i in xrange(self.N):
			for j in xrange(self.N):
				self.dist[i][j] = self.dist[j][i] = self.lpNorm(points[i], points[j], self.P)
		self.adj = self.getAdj()
		self.nxGraph = self.NXify()

	def lpNorm(self,A,B,p):
		x1, y1 = A
		x2, y2 = B
		return ((abs(x2 - x1)**p + abs(y2 - y1)**p)**(1.00/p))

	def getAdj(self):
		adj = dict()
		for i in xrange(self.N):
			for j in xrange(self.N):
				adj[(i,j)] = adj[(j,i)] = self.dist[i][j] <= self.W

		return adj

	def NXify(self):
		G = nx.Graph()
		G.add_nodes_from([x for x in xrange(self.N)])
		for i in xrange(self.N):
			for j in xrange(i,self.N):
				if self.adj[(i,j)]:
					G.add_edge(i,j)
		return G
```