Mini project report on Graph Embeddings

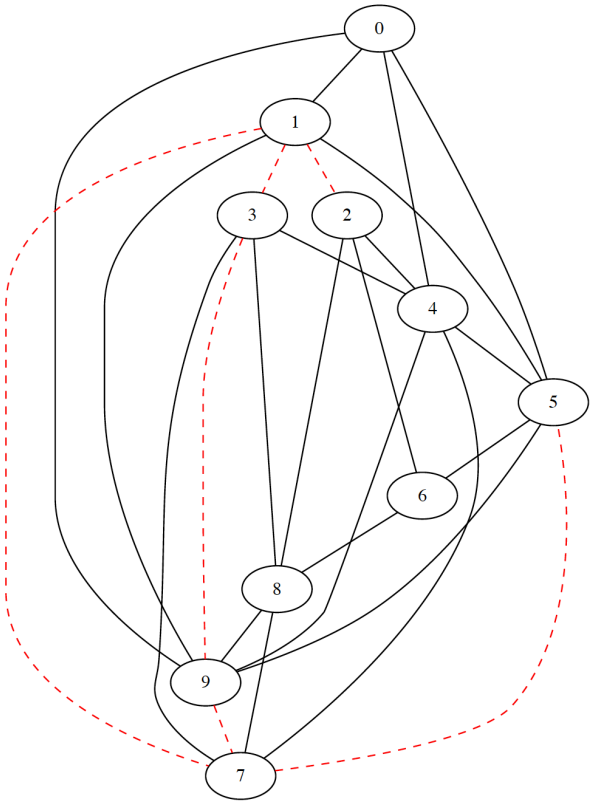
Participants:

Alon Bainer – 200718203

Shay Zilberman - 201735395

Group Advisor:

Professor Michael Elkin



Reference paper

Graph Spanners,
J. Graph Theory, 13 (1989), pp. 99—116
By D. Peleg and A. Schaffer

The algorithm

1. $Compute\ all\ clusters\ in\ G\ (the\ given\ graph)$:
   1.1: while number of uncovered neighbors of $cluster_i\ \geq n^{1/t} \times |cluster_i|$
   1.2**: $cluster_i = cluster_i \cup \{uncovered\ neighbors\ of\ cluster_i\}$
   1.3: $pick\ a\ new\ vertex\ from\ the\ uncovered\ vertices$
   $and\ start\ a\ new\ cluster = cluster_{i+1}$
2. $Build\ a\ new\ graph\ T\ s.t\ there\ is\ only\ at\ most\ one\ edge\ between\ two\ clusters$:
   $for\ every\ two\ cluster\ C_i, C_j\ in\ T, there\ exsit\ a\ maximum\ one\ edge\ (u,v)$
   $between\ them\ such\ that\ u \in C_i\ and\ v \in C_j\ and\ (u,v) \in |E|_G$

** - When 1.2 step of the algorithm is being done, we call this a "covering" action made by $Cluster_i$. The $uncovered\ neighbors\ of\ cluster_i$ are "being covered" by $Cluster_i$.

Implementation

We wrote the following python objects to support our algorithm:

1. Graph

```
class Graph_:
    def __init__(self):
    def addVertex(self,key):
    def getVertex(self,n):
    def __contains__(self,n):
    def addEdge(self,f,t):
    def getVertices(self):
    def __iter__(self):
```

2. Vertex

```
class Vertex:
    def __init__(self,key):
    def addNeighbor(self,nbr,weight=0):
    def __str__(self):
    def getConnections(self):
    def getId(self):
```

The graph is represented using adjacency list. Each vertex object has a list containing all of its neighbors, The function *getConnections()* returns that list. The function *addEdge(self,f,t)* adds f to t's adjacency list and adds t to f's adjacency list, where f,t are two vertex objects which had been created using the *__init__* constructor function.

3. Cluster

Actually, a cluster in our program is not implemented as a class object like the two above, but as an array in size of three. Where $Cluster_i[0]$ contains all vertices covered by $Cluster_i$, $Cluster_i[1]$ contains all vertices that are not covered by any cluster and have an edge with at least one vertex in $Cluster_i[0]$. $Cluster_i[2]$ stores the number of vertices in $Cluster_i[0]$.

cluster structure:

```
[[vertices of the cluster],[neigbors of all vertices in cluster],num of vertices in the cluster]
```

4. Global lists

- Global list L contains all clusters. Each entry in L represents a cluster structure as mentioned above.
- Global list *covered_vertices* contains all covered vertices of G. Variable *index* mentions the index of $Cluster_{index}$ in the list.

The program receives t,n parameters as arguments:

- t - represents the maximum distance t*|*path*| between every two vertices u and v in T , where *path* is the path between u and v in G.
- n - represents the number of vertices in G.

Creating the random graph G

We defined the density of G as a number p = 0.5. For each pair of vertices u, v, we generated a random number in the range of 0 to 1, and added an edge between them if that number was less or equal to p.

Creating the clusters

We used a recursive function in order to create the clusters, named *create_clusters()*. The function stop running when all vertices are covered, meaning every vertex is related to some cluster.

```
def create_clusters(graph,clusters,index,t,n,new_cluster_flag):
```

When *create_clusters() is* called, the function first checks if it should create a new cluster or keep maintaining the current cluster the function constructs. Checking the *new_cluster_flag* flag makes this decision: if it is set to 1 – start a new cluster, else (0) – keep working on the same cluster in list L. When first called from the main function, the flag is set to 1, stating that it should create the first cluster. The clusters are held in a global list L, s.t every entry in the list has a specific index. This index matches the index of the corresponding cluster.

When creating a new cluster $Cluster_i$, it is stored in the ith place in list L. The function picks the next vertex v_j (according to its index in the vertices list of G), which is not covered by any cluster created so far by the algorithm. The algorithm adds v_j to $Cluster_i[0]$, $Cluster_i[1]$ and to *covered_vertices* list *as well.* $Cluster_i[2]$ is set to 1.

When maintaining a cluster that created in previous steps of the algorithm (*new_cluster_flag* = 0), the algorithm will not choose a new vertex to start a new cluster. Instead, it will work on the same cluster in L indexed by *index*. The function receives as an argument the index of the cluster in L, which is currently being built.

Next, in both cases, the algorithm iterates over the vertices in $Cluster_i[0]$. It adds all vertices that are neighboring vertices in $Cluster_i[0]$ and are not yet covered by any cluster, to a list called *found_on_this_level.* At this point, the algorithm checks if the number of vertices in f*ound_on_this_level* is bigger or equal to n^(1/t)* <number of vertices currently in the cluster> .

If the condition holds, we add all vertices in *found_on_this_level to* $Cluster_i[0]$ (step 1.2), and replace all vertices in $Cluster_i[1]$ with them. $Cluster_i[2]$ is increased by $|found\_on\_this\_level|$. The algorithm marks all vertices in *found_on_this_level* as covered by inserting them to *covered_vertices* list. Finally, the function recursively calls itself with new_cluster_flag = 0 and *index* remains the same.

Else, i.e. the condition doesn't hold, *index* is increased by 1, and the function is recursively called with new_cluster_flag = 1.

Intuitively, in each recursive iteration of the *create_clusters()* function we hold at $Cluster_i[1]$ all vertices that were added to the cluster at the previous iteration. We do so In order to find their neighbors in the current iteration, which are the only potential vertices that can be covered the cluster at this point.

Finding connections between clusters

Finishing the previous part leaves us a list of a list of clusters L. There is no information about any edges between vertices or clusters. This part shows how clusters are connected to each other in the newly constructed graph T:

At first, we use *find_edge_between_clusters()* function. The function finds for each pair of clusters, C_i, C_j, all pairs of vertices u,v s.t vertex u is covered by C_i, vertex v is covered by C_j, and (u,v) is in $|E|_G$, then it stores them as tuples in a list. Next, we maintain an K*K matrix, where k is the number of clusters. In each entry [i,j] ,which representing adjacency between C_i, C_j, we store the first pair of vertices u,v from the list above. In case two vertices from different clusters C_i, C_j have an edge connecting them in G, their corresponding clusters should have an edge connecting them (and only one) – this edge is represented by a tuple of two vertices, and should be placed in the matrix in [i,j] cell.

At this point we have a list of clusters, and a matrix of clusters, stating which of the clusters should be connected by an edge, and which are the vertices that should be connected in order to connect those clusters. If two clusters shouldn't be connected, their corresponding entry in the matrix is assigned with a zero.

Constructing the T-spanner

At this stage, the algorithm creates a new graph T. the vertices are being copied from G without their corresponding adjacency list. The edges are sub-group of $|E|_G$, and haven't been added yet.

The first edges added to $|E|_T$ are between vertices of the same cluster. The algorithm iterates over the vertices list of G (G.vertList). For each vertex u_i, the algorithm iterates over all of it's neighbors, and adds an edge u_i, v (where v is some neighbor) to $|E|_T$ iff u_i and v belong to the same cluster.

The next step is creating edges between clusters. The algorithm iterates over the matrix mentioned in the previous step, and adds to $|E|_T$ all edges found in the matrix entries.

Experiments

Experiment #1:

input:

- n=300 (number of vertices)
- t=5

Measuring the average number of edges which had been removed from the original graph when creating the t-spanner graph, as a function of graph-density p.

p varies in range of 0.05 to 0.5 by steps of 0.05.

For each density we create 10 random graphs. We accumulate the number of edges that were removed at each execution, and display the average.

Experiment #2:

input:

- n=200 (number of vertices)
- p=0.05 (graph density)

Measuring the average number of edges which had been removed from the original graph, when creating the t-spanner, as a function of the t parameter.

t varies in range of 1 to 10 by steps of 1.

For each t parameter we create 10 random graphs. We accumulate the number of edges that were removed at each execution, and display the average.

Experiment #3:

input:

- t=20
- p=0.05 (graph density)

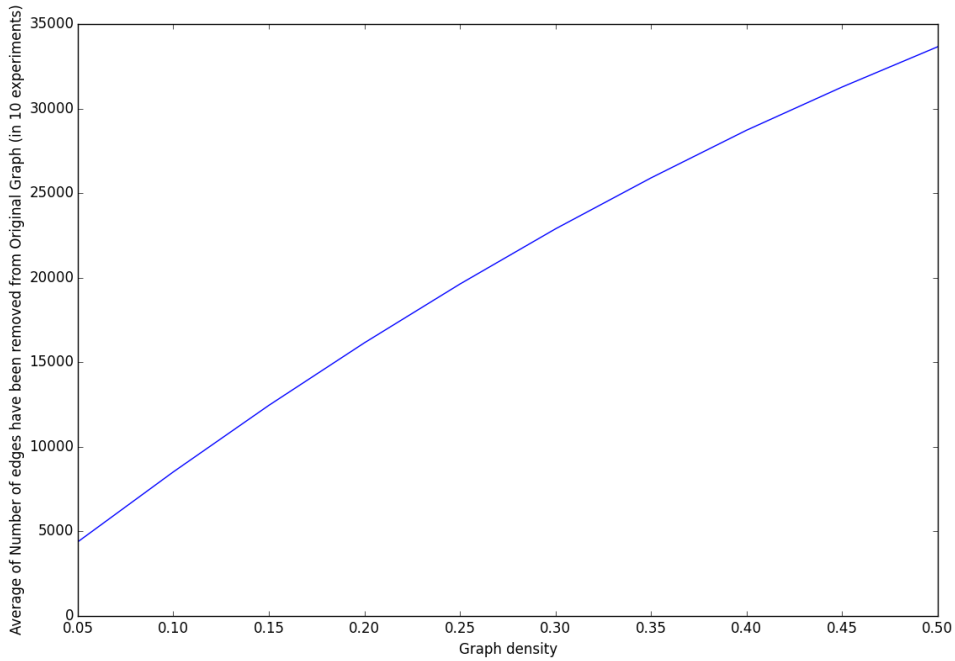
Measuring the average number of edges which had been removed from the original graph when creating the t-spanner, as a function of number of vertices n.

n varies in range of 20 to 200 by steps of 20.

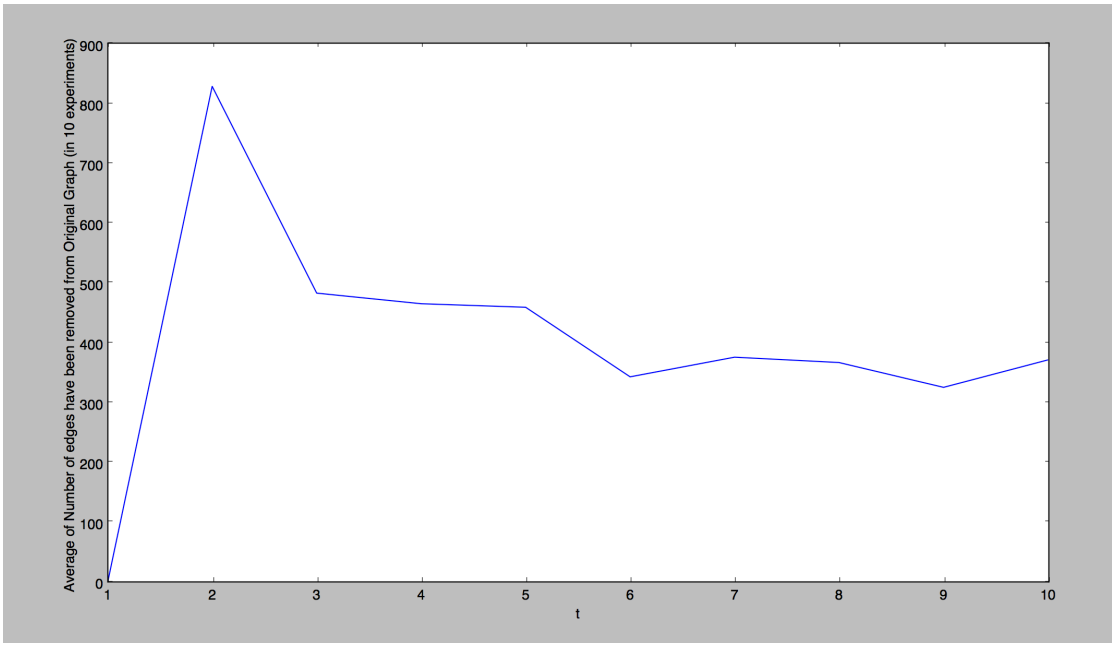
For each n parameter we create 10 random graphs. We accumulate the number of edges that were removed at each execution, and display the average.

Results

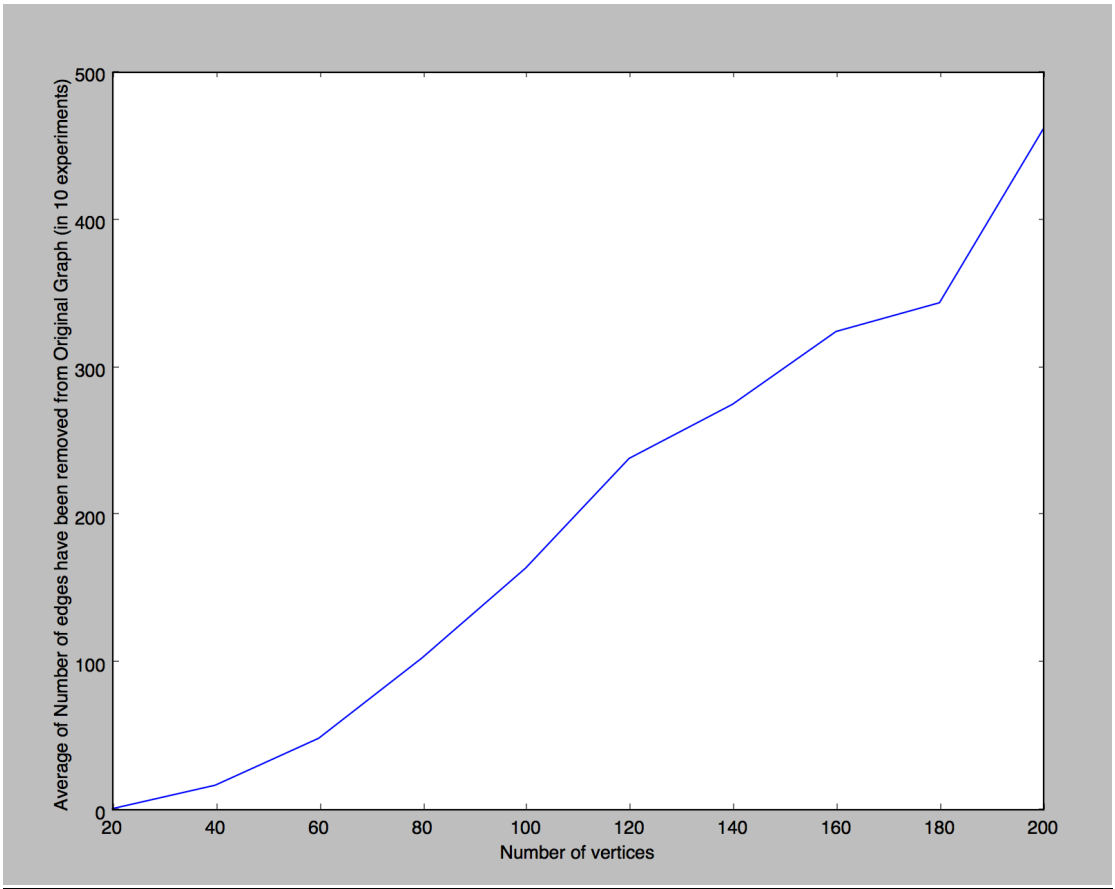
Experiment #1:



Experiment #2:



Experiment #3:



Validation tests

As a part the project, we did some tests in order to examine the correctness of the algorithm. Right after the t-spanner graph is built, the function *Tests()* is being called. The function executes two different tests:

1) Epsilon test:

$|E|_C \leq n^{1+\frac{1}{t}}$. Where $|E|_C$ is number of total edges connecting the clusters.

2) Radius test

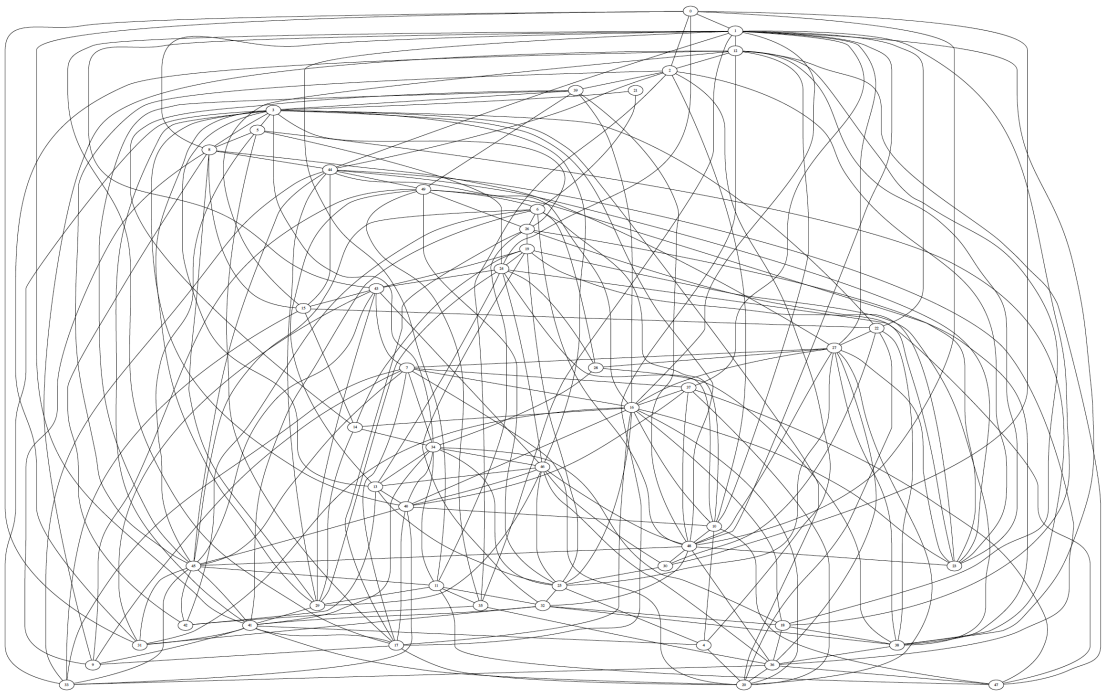
For each cluster we checked that its radius is $\leq t-1$. For each cluster, we computed the distance from each vertex covered by the cluster, to the center of the cluster (center = the first vertex that had been covered by the cluster).

Visualization

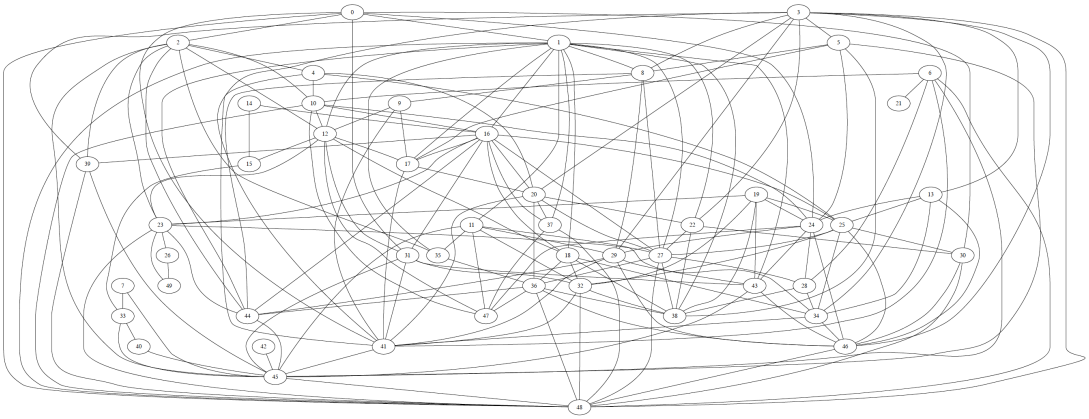
The program provides a visualization of a data base graph using the graphviz python library. When constructing the random graph G at the beginning of the program, the visual graph is being built as well. The t-spanner graph T, which is being built at the later part of the program, is being built simultaneously using graphviz library.
Finally, the program builds a “comparison graph”. In the comparison graph, the edges that had been removed from G when constructing T are shown in dashed red lines.

for example, let us take a look at the visualization output for a given input n=50, t=10, p (graph density) = 0.5:

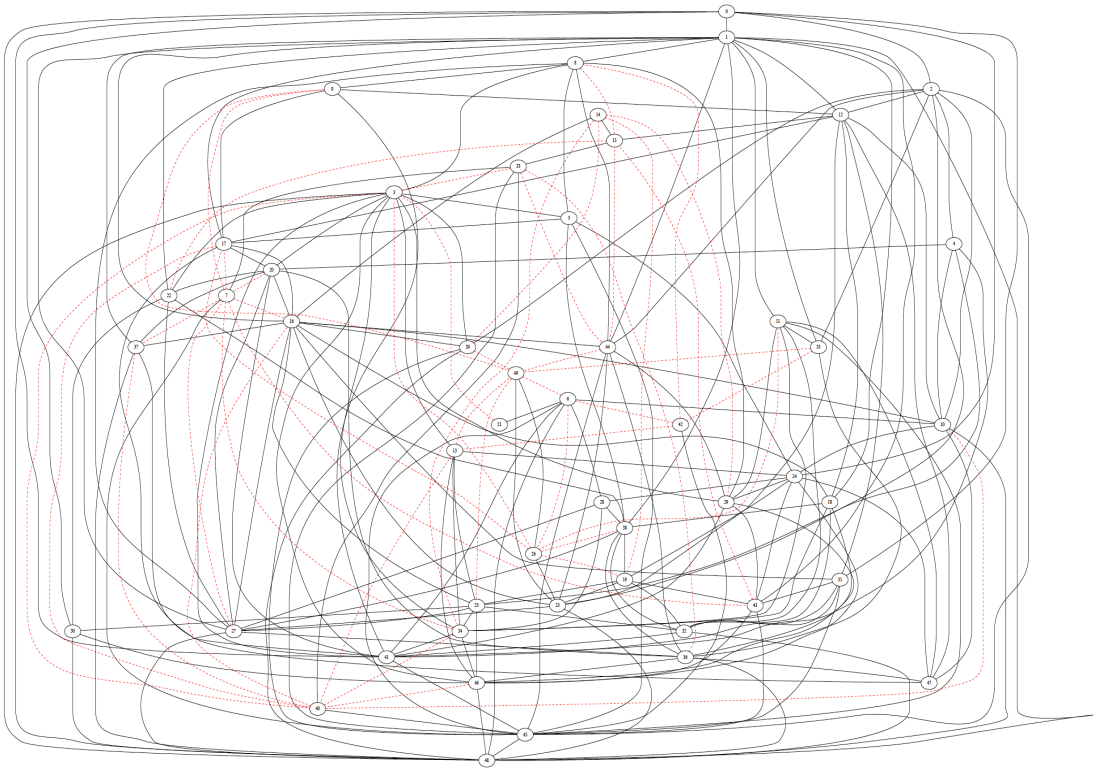
original graph G:



t-spanner graph T:



Comparison graph:



Conclusions

Our goal was to explore the properties of the t-spanner of a given graph G. The whole process was very enriching, we made a lot of educating mistakes, but finally succeeded and we are very proud to present our project's report.

After we finished the implementation, we made few tests in order to check the algorithm. Then, we did some interesting experiments. The results helped us further understand how t-spanners behave in different environments. Python offers plenty of support for graphs (as graphviz library) and it was very challenging to learn how to use it.

Our main conclusion from the project is that t-spanners may be practical. By removing connections between computers there will be reduction in the inter-communications resources required, while the length of the path of a message from one computer to another will be increased by a constant factor equals to 't'.

Conclusions from the experiments:

Exp 1:
The total number of edges being removed when applying the t-spanner algorithm is proportional to the density of the graph. The increasing of the density cause an increase in the number of removed edges with an approximately linear relation.

Exp 2:
As t increases, the number of edges that had been removed from G is getting smaller, but this property isn't absolute as the other experiments. When t equals to one, the number of removed edges is obviously equals to 0, because the meaning of t=1 is that the maximum distance t*|*path*| between every two vertices u and v in T (where *path* is the path between u and v in G) is actually 1*|*path*| = |*path*|, so none of the edges can be removed.

Exp 3:
It can be seen that when n=0, the number of edges that had been removed from G while constructing T equals to 0, since zero vertices leads to zero edges. The total number of edges being removed when applying the t-spanner algorithm is proportional to the number of vertices of the graph.