Mini Project On-

$(1+\varepsilon,\beta)$ −Spanner Constructions for General Graphs

Michael Elkin ∗ David Peleg ∗

December 25, 2002

By:

Kathrine Smoliansky & Ronen Finish

Supervisor:

Prof. Michael Elkin

Table of content-

Introduction-

This paper describes our journey of studying and applying the algorithm presented in the paper- "$(1+\varepsilon,\beta)$ −Spanner Constructions for General Graphs", by Michael Elkin and David Peleg.

A spanner is a subgraph H of an original graph $G=(V,E)$, such that H has fewer edges than G, whilst maintaining the locality properties of the network. We measure this locality properties using a "stretch" factor, namely the worst multiplicative factor by which distances between a pair of vertices is increased as result of using the spanner edges alone.

We studied in a "Metric Graph Algorithms" course by Elkin Michael, that for an integer parameter k, a stretch of $O(k)$ can be guaranteed by a spanner using $O(n^{1+\left(\frac{1}{k}\right)})$ edges, namely a $k-spanner$. This construction has a great meaning for network routing as the network becomes much smaller, but the distances are maintained within a acceptable scale, and routing becomes faster.

In this paper the innovative idea that we studied is the integration of an additive factor to the stretch, the addition of sufficiently large constant allows the multiplicative factor to be arbitrarily close to 1.

A $(\alpha,\beta)-spanner$ is a subgraph H of $G=(V,E)$ if- $dist_H(u,w)\leq\alpha\cdot dist_G(u,w)+\beta$ for every $u,w\in V$. Were $\alpha=1+\varepsilon$ and $\beta=k^{loglogk-log\varepsilon}$.

Algorithm Sp Cons-

The spanner constructing algorithm uses two routines: Procedure Down Part and Procedure SC. The algorithm receives a graph and two parameters: kappa and epsilon and calculates the parameters : $J = \log(k)$, $\delta_j = \left(\frac{\log(k)}{\varepsilon}\right)^{\mathrm{j}}$, $\sigma_j = n^{t_{J-j}-t_{J-j+1}}$, $t_j = \frac{(\kappa - 2^{j-1})}{(2^{j-1}\kappa)}$,

$\tau_j = \left[\kappa t_{J+1-j}, \kappa t_{J-j}\right)$ $(i.e. \tau_1 = [1,2,3), \tau_2 = [3,4,5,6,7)\ ect..)$.

First, the algorithm uses *Procedure Down Part* to create a basic spanned partition including the center vertex for each cluster, and a spanning tree. The procedure simultaneously creates a sub-graph consisting of spanning trees of all the clusters and shells.

The algorithm then iterates over all the clusters, created in the last step, by their radii and creates super clusters, consisting of smaller clusters which at most are $2\delta_j$ apart, using the *Procedure SC*. Upon returning from the procedure SC, it adds the spanning trees of the new clusters to the spanner, collects all the partitions to C' and collects the left-out clusters to R. In the end we are left with spanned partitions and clusters that were left out. The last step is to calculate the shortest path between the ones who are $2\delta_J$ apart from each other and to add them to the spanner too.

The algorithm then returns the spanner.

Input: Graph $G = (V, E)$, integers κ, ε.
Output: $Subgraph\ H$
1. $(\mathcal{G}, H) \leftarrow Down\ Part(G)$;
2. $C' \leftarrow \emptyset$;
3. $For\ j = 1\ to\ J - 1\ do$:
(a) $C \leftarrow C' \cup S_i \in \tau_j\ A_i(\mathcal{G})$;
(b) $(C'', H', R') \leftarrow SC(G, C, \sigma_j, 2\delta_j\ (Y, J))$;
(c) $C' \leftarrow C' \cup C''$; $R \leftarrow R \cup R'$;
(d) $E(H) \leftarrow E(H) \cup E(H')$;
(e) $j \leftarrow j + 1$;

4.$\mathrm{R} \leftarrow \mathrm{R} \cup \mathrm{C}' \cup \mathrm{S_i} \in \tau_\mathrm{J}\ \mathrm{A_i}(\mathcal{G})$;

5. ***For*** $every\ pair\ of\ spanned\ clusters\ (v_i, S_i, T_i), (v_j, S_j, T_j) \in R$
$such\ that\ distG(S_i, S_j) \leq 2\delta_J\ (Y, J)\ do$:
(a)Calculate the shortest path $\mathrm{P_{ij}}$ between $\mathrm{S_i}$ and $\mathrm{S_j}$;
(b)$\mathrm{E(H)} \leftarrow \mathrm{E(H)} \cup \mathrm{E(P_{ij})}$.

Procedure Down Part-

This procedure receives a graph G. It initializes a group of vertices U that includes all the vertices from G, an empty spanned partition $\mathcal{g}$, centered shells S', and a subgraph consisting of spanned trees $\mathrm{E(H)}$.

In a loop until there are no vertices left, we pick one edge from U, and mark it as the center of the cluster S that we are about to create. We collect all the neighbors of S in phases until we reach the phase in which the number of neighbors of S that are still in U is $n^{\frac{1}{\kappa}}$ times smaller than the size of the spanner that we have already collected. Then we take S and all its neighbors (who weren't included in S) and we push them all into the shell $\widehat{S}$.

Now that we have our cluster S we push it into the spanned partition which is a set of triples: <S- the cluster, v- its center and T the spanning tree of S>. We also collect S' couples of <shells $\widehat{S}$ with their center v>, and we remove from U all the vertices that were collected into the cluster S.

- Note- there might be also clusters consisting of only one vertex if it has too few neighbors.

When there are no vertices left in U, i.e., all the clusters are constructed, the procedure iterates over all the shells that were collected into S' and creates spanning trees for them, adding each one to H. The procedure then returns the spanned partition and H.

Input: Graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$.
Output: Spanned partition g, subgraph H.
1. $\mathrm{U} \leftarrow \mathrm{V}$; $\mathcal{g} \leftarrow \emptyset$; $\mathrm{S}' \leftarrow \emptyset$; $\mathrm{E(H)} \leftarrow \emptyset$;
2. ***while*** $\mathrm{U} \neq \emptyset$ do:
(a) Pick an arbitrary vertex $v \in \mathrm{U}$;
(b) $\mathrm{S} \leftarrow \{\mathrm{v}\}$;
(c) ***While*** $|S \cup \Gamma(S) \cap U| \geq n^{\frac{1}{\kappa}}|S|$ *do*:
$A.\ \mathrm{S} \leftarrow \mathrm{S} \cup \Gamma(\mathrm{S}) \cap \mathrm{U}$; the cluster
(d) $\widehat{S} \leftarrow S \cup \Gamma(S) \cap U$; *the shell*
(e) Create a spanning tree T for S;
(f) $S' \leftarrow S' \cup \{(v, \hat{S})\}; \mathcal{g} \leftarrow \mathcal{g} \cup \{(v, S, T)\}; U \leftarrow U \setminus S$;
3. ***For*** *every* $(v, \hat{S}) \in S'$ *do*:
(a) *Create a BFS spanning tree* T' *rooted at* v *for* $\hat{S}$
$(namely, a\ tree\ spanning\ \hat{S}\ and\ yielding\ shortest\ paths\ to\ v\ in\ the\ induced\ subgraph\ G(\hat{S}))$;
(b) $E(H) \leftarrow E(H) \cup E(T')$;
4.Return($\mathcal{g}$, H);

Procedure SC-

This procedure receives a graph G, spanned partition C, and the parameters sigma and delta. It initializes U with all the spanned clusters from C, the subgraph H' and spanned partition of all the super clusters C'.

In a loop until there are no spanned clusters S left in U, such that the number of delta neighbors of S that are still in U is smaller than sigma, we pick one cluster S from U and mark its center vertex v as the center of the super cluster S' that we are about to create. We pick one cluster S from U per iteration and collect all the delta neighbors S_i of S. For each cluster S_i that we add to S' we also add its spanning tree to T', and the shortest path between S_i and S to T'.

Now that we have our super cluster S' we push it into the super spanned partition C' which is a set of triples: <S'- the super cluster, v- its center and T' the spanning tree of S'>. We remove from U all the clusters that were collected into the super cluster S', and we collect T' to the subgraph H'.

The clusters left in U are now transferred to R. For every pair of clusters in R that are at most a delta apart we calculate the shortest path P between them and add P to H'.

The procedure then returns C', H' and R.

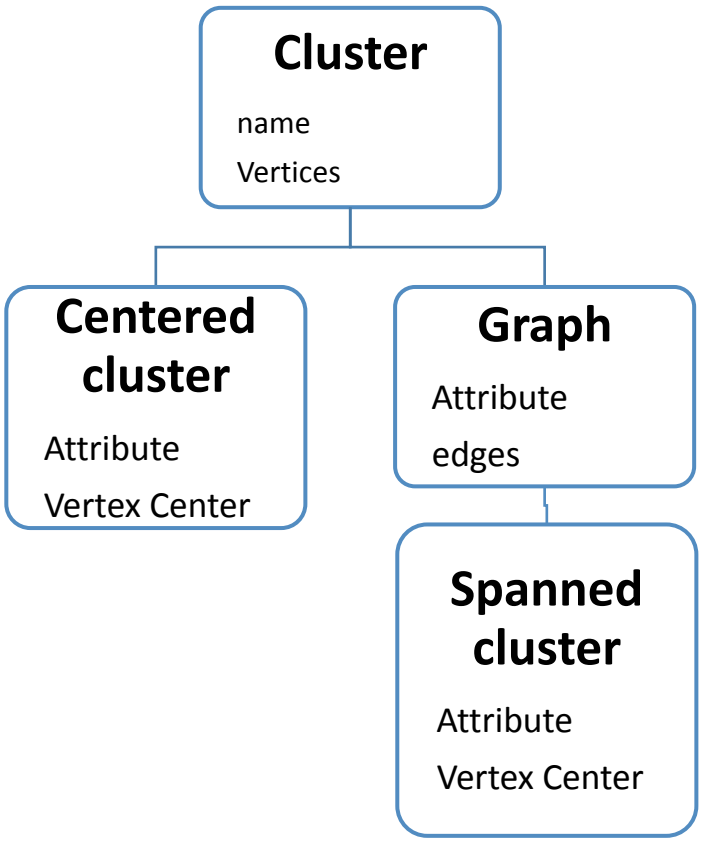
Input: Graph $G = (V, E)$, spanned partition $C = \{(v_i, S_i, T_i)\}$ of G, integers $\sigma, \delta \geq 1$
Output:
Spanned partition $C' = \{(v_i', S_i', T_i')\}$ *of* G, *subgraph* H', *collection* R *of unmerged clusters.*

1. $E(H') \leftarrow \emptyset$; $U \leftarrow C$; $C' \leftarrow \emptyset$;
2. ***while*** *there exists a spanned cluster* $(v, S, T) \in U$ *such that* $|\Gamma_u(S) \cap U| \geq \sigma$ *do*:
   (a) $S' \leftarrow \bigcup_{(v_i, S_i, T_i) \in \Gamma U(S)} S_i$;
   (b) i. $T' \leftarrow T$;
   ii. **For** every S_i such that $(v_i, S_i, T_i) \in \Gamma_u(S)$ do:
   A. Compute the shortest path P_i in G between the clusters S and S_i ;
   B. $E(T') \leftarrow E(T') \cup E(P_i) \cup E(T_i)$;
   (c) $C' \leftarrow C' \cup \{(v, S', T')\}$;
   (d) $U \leftarrow U \setminus \Gamma_u(S)$;
   (e) $E(H') \leftarrow E(H') \cup E(T')$;
3. $R \leftarrow U$; Remaining (unmerged)clusters
4. ***For*** *every pair of spanned clusters* $(v_i, S_i, T_i), (v_j, S_j, T_j) \in$

Our Implementation (JAVA):

We chose to implement this algorithm in java because it has many objects, classes and libraries that had to be constructed which are simply implemented in an object-oriented language such as JAVA.

The hierarchy is described in a tree:



Selected implemented methods:

SpCons class-

In this section we will present an overview of the functions that are part of the **SpCons** class. This class includes only the implementation of the algorithm itself and some methods that are necessary for some calculations.

public static Graph SpCons(Graph G, int k, double _epsilon);

The SpCons method is the main part of the algorithm which constructs the spanner. It first calculates the spanned partition using the downPart procedure. Then it iterates over the j values, and in each iteration it collects the super clusters with increasing radius value using procedure SC. Lastly, it goes over all the clusters which weren't added to the super clusters

and finds the shortest path for each pair that is at most 2 deltas apart, returning the spanner consisting of all the shortest paths and the spanning trees of the clusters.

public static Graph downPart (Graph G, Set<SpannedCluster> partitionG);

The downPart method receives as parameters the graph G and the partition. In JAVA, a function cannot return more than one object. Therefore, it returns the subgraph and it feeds the partition into the object it receives as parameter. It goes over all the vertices in the graph and creates clusters out of them with regard of the number of the neighbors they have. The idea is to collect the high-density areas into clusters.

public static Graph ProcedureSC(Graph G,Set<SpannedCluster> partitionC, int sigma, double delta, Set<SpannedCluster> partitionCTag, Set<SpannedCluster> partitionR);

This procedure returns a subgraph. It receives as parameters the Graph G, the partition from the previous procedure, sigma and delta parameters, the object for the new partition to go into and an object that will save all the clusters that weren't incorporated into other clusters. The procedure SC creates super spanned clusters out of smaller clusters with increasing radii. And returns a subgraph of spanning trees and shortest paths.

private static Set<SuperVertex> expandNeighbors (Cluster s, Cluster vertexSetU, Graph G);

This method receives three arguments: the cluster that we are interested in expanding, the cluster of all the available vertices (vertices that are not part of some other cluster), and the graph G. It uses the getNeighbors method of the class Graph and then makes sure that all the vertices are available and returns a new cluster with the neighbors of cluster S.

public static int calcNeighbors(Cluster s, Cluster vertexSetU, Graph G);

This method uses the previous method to count the number of neighbors.

public static Set<SpannedCluster> getAi(Set<SpannedCluster> g, int j);

This method returns the neighbors that have a specific radius.

Graph class-

The following section will list some interesting methods that we implemented in the class **Graph**. This class has close to 40 methods, the majority of which are very basic: getters, setters, methods that remove, insert, etc... We will now briefly explain a few.

public Set<SuperVertex> getNeighbors(Set<SuperVertex> rv);

This method receives a set of vertices and for each one of them it uses another method with the same name that finds all the neighbors of a vertex. The method iterates over all the edges that are connected to the vertex and collects all the vertices on the other side of the edge.

public int getSPTForUnWeightGraph(SuperVertex sourceVertex);

This method receives the root of the tree that we are about to construct and uses a recursive method which in each iteration adds an edge until it covers all the vertices. It returns the number of edges in the tree.

public Set<SuperVertex> getLNeighbors (Set<SuperVertex> vertices, double l);

This method returns all the neighbors for a set of vertices that are at most l distant from each other.

public Graph getShortestPath (SpannedCluster sourceCluster, SpannedCluster targetCluster);

This method calculates the shortest path between two clusters. It goes over every couple of vertices (one from each cluster) and chooses the shortest path of all the options.

Experiments-

Our goal is to see how different Kappa values effect the ratio between the number of edges before and after the spanner calculation as well as how it effects the distances between the vertices i.e. the stretch.

The main challenge is generating meaningful graphs, the main reason is that this algorithm is aimed toward very big and difficult graphs, but it is not possible to run the algorithm experiments on graphs with more than 200 vertices (generating hundreds of tests), and it's almost impossible achieving radius that is bigger than 5 generating random edges and keeping the graph with meaningful density.

Our solution is generating random groups of vertices with probabilities varies in the range [0.2-1.0] in 0.2 jumps and afterwards connecting the vertices between the groups in probability range of [0.01-0.09] in 0.02 jumps. For each graph we randomly selected ten vertices and measured the shortest distance between each pair in the original graph as well as in the spanner. We generated 5 times each experiment, so we assure statistical significance and calculated the average, maximum and minimum difference in distance, and the ratio between the number of edges in the original graph and the spanner.

The parameters we applied on the algorithm are $\varepsilon = 0.5$ and $kappa[3-7]$. We generated total of 625 spanners, (25 graphs with different probability, for each we calculated five spanners with different factors, we generated each experiment five times for statistic accuracy). To analyze the algorithms behavior. We displayed the results in two types of graphs, one shows the change in the number of edges and the other shows the change in distance.

- Figure 1- Comparing the number of edges in the original graph and the spanners with different kappa value for each graph probability.
- Figure 2- Comparing the proportion of original graph to spanner number of edges with different kappa values for each graph probability.
- Figure 3- Each curve represents the average distance in the spanner with different kappa values.
- Figure 4- Each curve represents the proportion in distance of the spanner and the original graph with different kappa values.
- Figure 5- Each curve represents the average minimal distance in the original graph and spanner with different kappa values.
- Figure 6- Each curve represents the average maximal distance in the original graph and spanner with d
- Different kappa values.

Analyzes & Conclusions:

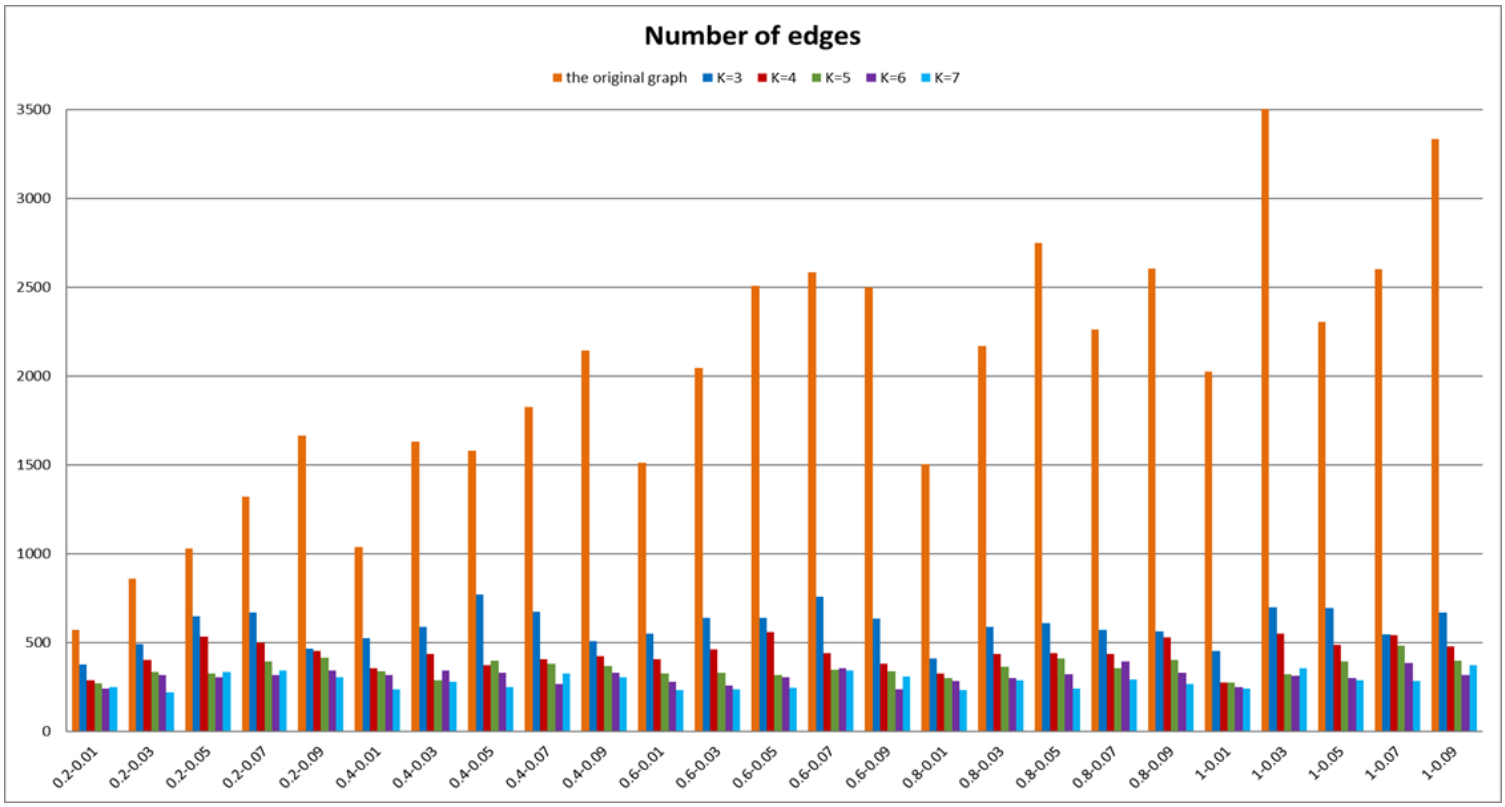


Figure 1

Figure 1 represents the average number of edges of all the graphs and their spanners from our experiments, for each probability we generated 5 instances of the graph for better statistical significance and for each one of the graphs we calculated the spanners with different kappa factor and we calculated the average number of edges.

We observe from figure 1 a general trend of increase in the number of edges throw-out the experiment but the size of the spanners has similar scale. Also, there are a few statistical deviations that probably accure because we recited only five times each experiment and its not enough to avoid statistical deviation.

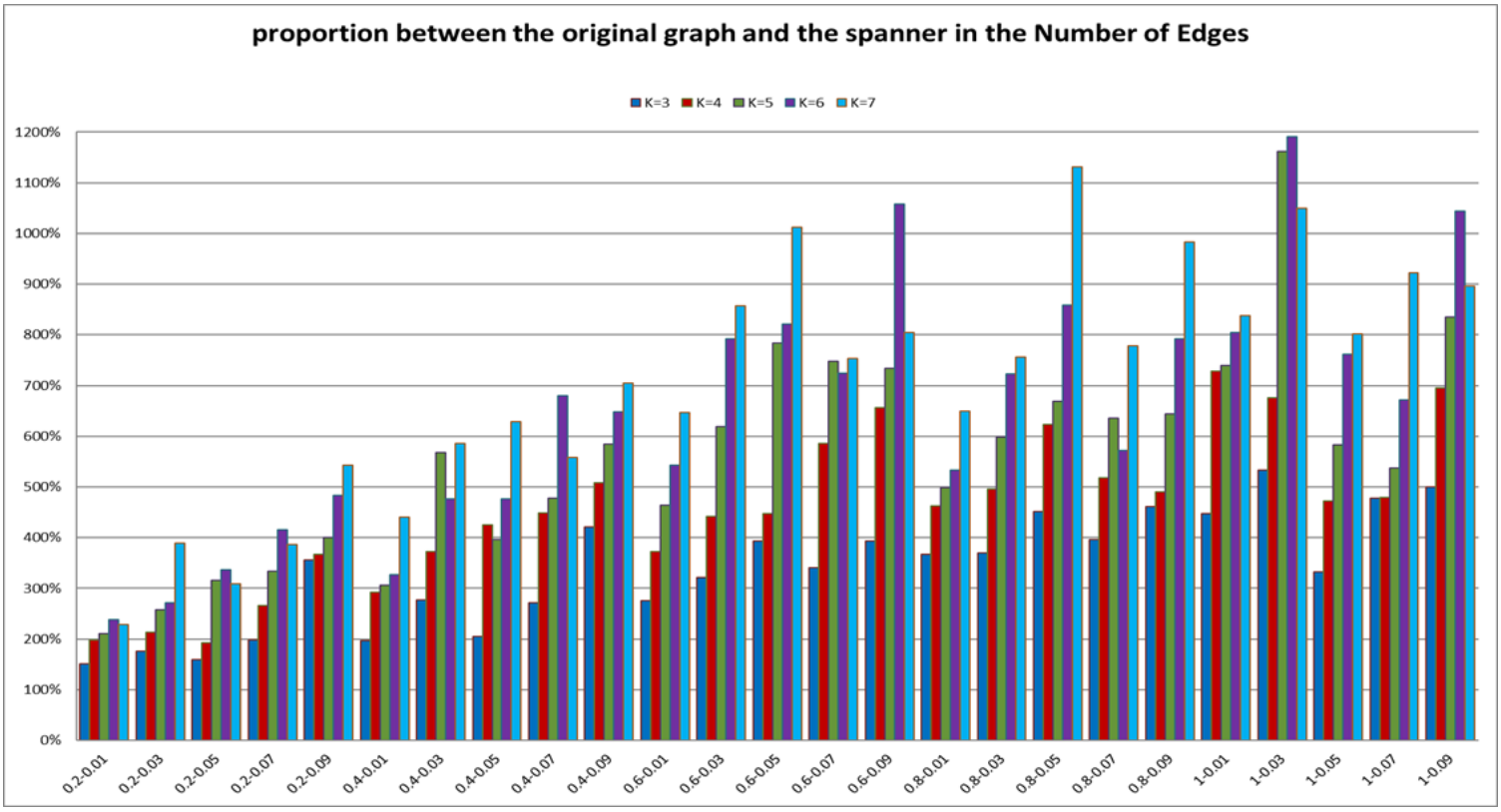


Figure 2

Figure 2 represents the average proportion of $\frac{number\ of\ edges\ in\ the\ graph}{number\ of\ edges\ in\ the\ spanner}$ from our experiments, for each probability we generated 5 instances of the graph for better statistical significance and for each one of the graphs we calculated the spanners with different kappa factor and we calculated the average proportions. We observe a steady increase in the proportions because the number of edges in the original graph increases but as we mentioned before the scale of the spanner is quite similar

As seen in figure 1 we can observe the statistical deviations.

It is clear from figure 1 and figure 2 that the difference, in the number of edges, between the original graph and the spanner is increased when the density of the graph increases. The most significant improvement in the size of the spanner regarding the factor kappa is between the values 3 and 4 although it increases with higher values of kappa.

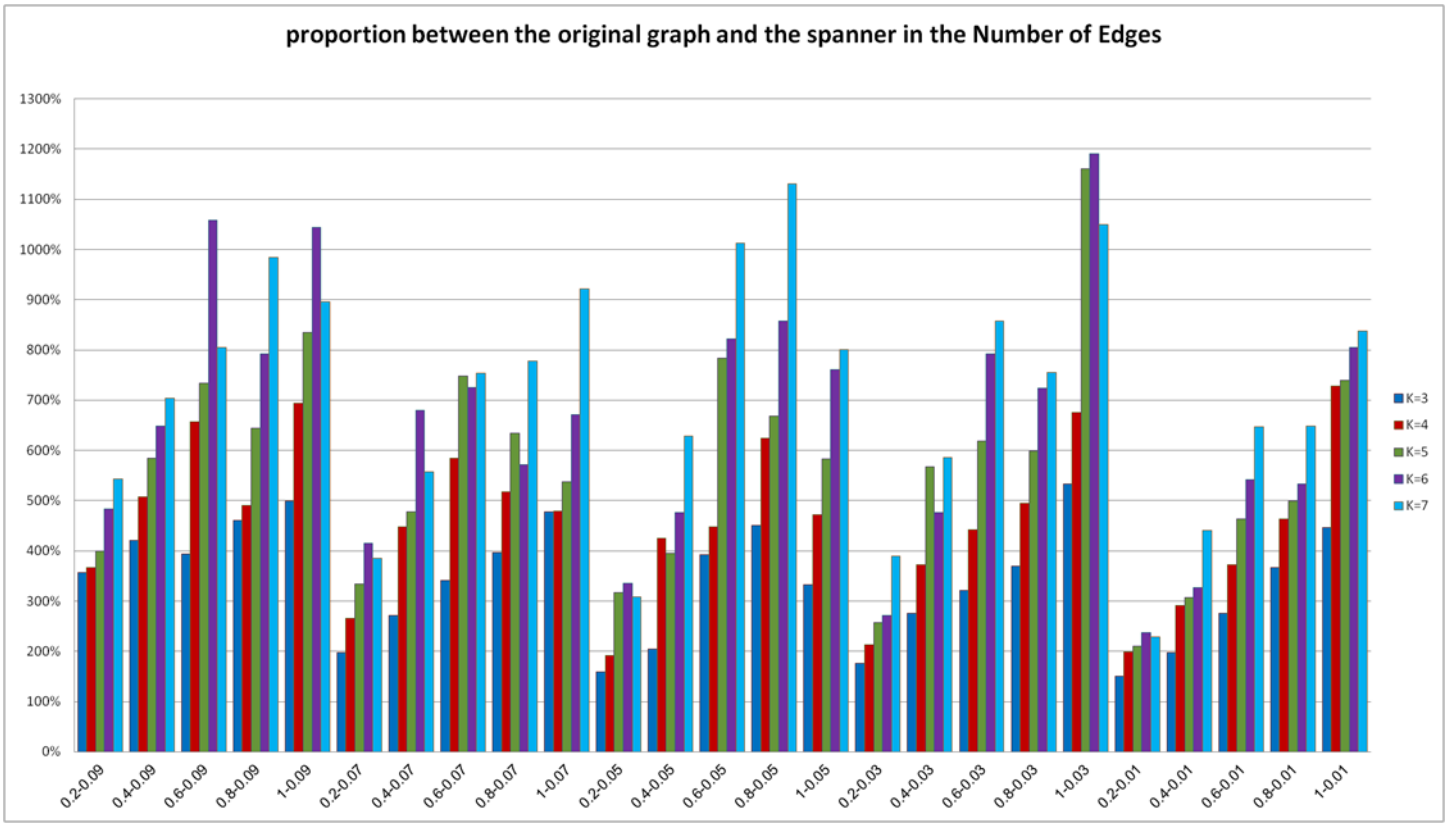


Figure 2.1

Figure 2.1 is generated from the same data as figure 2 but sorted differently.
It is more visible from figure 2.1 that the gap, in the number of edges, between the original graph and the spanner is increased when the density of edges in the sub-graphs increases.

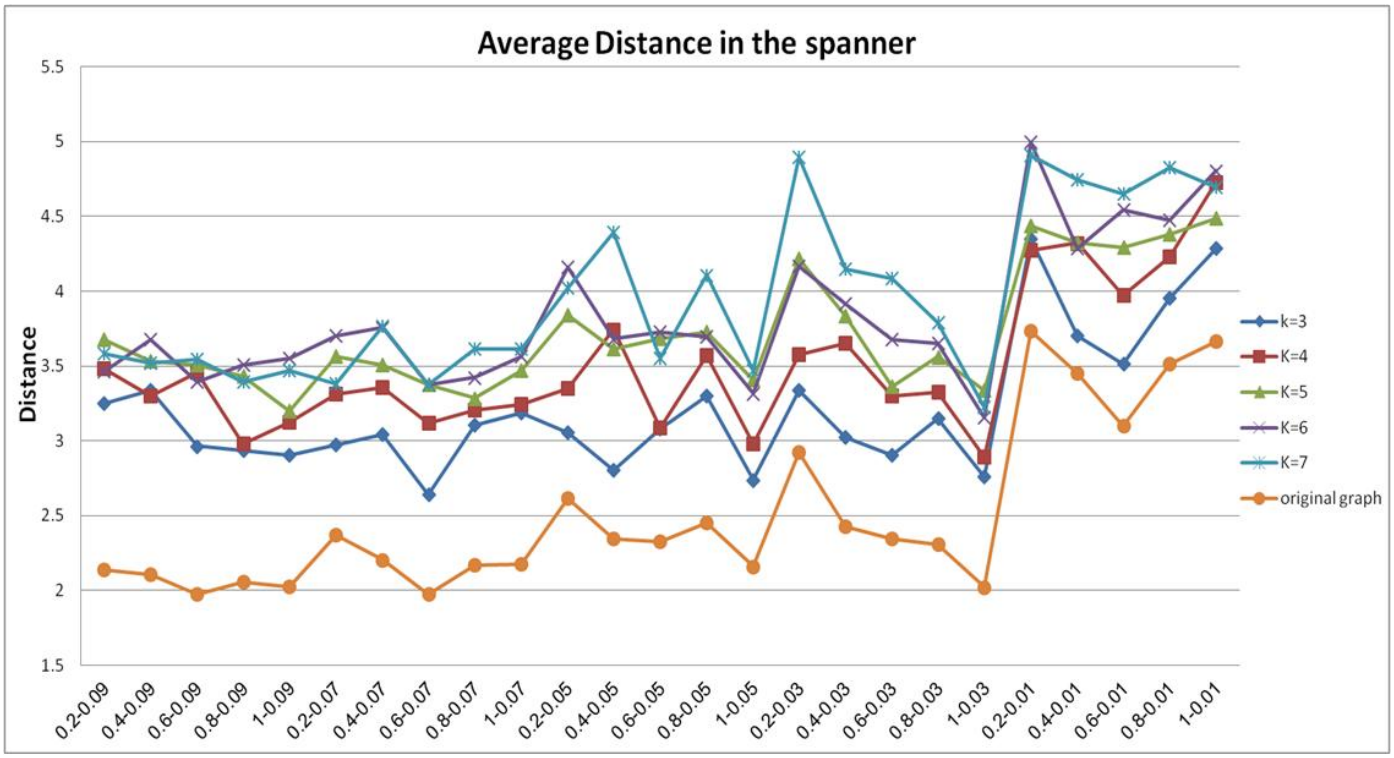


Figure 3

In figure 3 we measured the average distance in the graph and in the spanner, for each probability we generated 5 instances of the graph for better statistical significance and for each one of the graphs we calculated the spanners with different kappa factor and we calculated the average distances. We see that the average distances have a similar behavior in proportion to the original graph.

From figure 3 we can learn that the less edges connecting between sub-graphs, the average distance between the vertices in the graph is bigger. And in general the probability of edges between sub graphs has more significant influence on the average distances in the graph.

Conclusions:

It is clear from figure 1 and figure 2 that the difference, in the number of edges, between the original graph and the spanner is increased when the density of the graph increases. The most significant improvement in the size of the spanner regarding the factor kappa is between the values 3 and 4 although it increases with higher values of kappa.

From figures 3 we learn that distance approximation is best at kappa 3 and gradually the occurrence decreases until it reaches the worst approximation with kappa 7.

In figure 4 we can see that although the kappa 3 factor gives the best approximation for distances, the spanners never reaches double the distance from the original graph. Overall, it's clear that the difference in distances between the original graph and the spanners is 1.5 on average and it corresponds with our expectations for the additive and multiplicative factors.

The upshot from form figure 5 is straight forward, when the edges that connect between clusters are denser the distances in the graph are shorter.

Figure 6 is not making much sense by itself except the obvious conclusion that the shortest distances are in the original graph. But when looking at both figure 5, 4 and 6 the conclusion is that the spanner shows the best results for the maximal and average distances because of the low multiplicative factor but for the minimal distances the results are not conclusive because of the relatively big additive factor.

Bibliography:

1. $(1+\varepsilon,\beta)$ −Spanner Constructions for General Graphs.

December 25, 2002 by Michael Elkin and David Peleg

2. Distributed Deterministic Construction of Sparse Near-Additive Spanners

January 14, 2019 by Shaked Matar and Michael Elikn.

Appendices- Excel results and JAVA code:

The link for the Excel documentation of our experiments results are in the fallowing link in google sheets:

https://docs.google.com/spreadsheets/d/1d5bLfy4WOkwYtLO50cxj-5jhX8rK56WZ8VdywAEyXlU/edit?usp=sharing

Our full implementation including all the Graph and spanner classes we created to support our program is in the fallowing link in GitHub:

https://github.com/ronen0072/Mini-Project-on-Graph