T-Spanner Algo: using Sen-Baswana

0. Introduction:

The project is to implement an algorithm which generates a t-spanner. A t-spanner of a graph is a subgraph containing all the vertices but only a subset of the edges, such that the distance between two vertices in the t-spanner is at most t times the distance between the same vertices in the original graph. The t in $t - spanner$ is sometimes written as $2k - 1$ as we typically work with odd values of t. So, t-spanners are sometimes called $2k - 1$ spanners.

Why do we need t-spanners?
When we want to find distance between any two points in a graph, what we can do is use Floyd-Warshall and get all pairs shortest paths in $O(n^3)$ time for dense graphs where n is the number of vertices. But this might be too much when accuracy is not very important. The t-spanner algorithm produces a graph where the number of edges are less than the order of m (number of edges in a dense graph). This improves the time to compute all pairs shortest as it will be less than $O(n^3)$. Although this comes with a tradeoff in accuracy.

1. The Beginning

Our project is based on the research paper by Sen and Baswana [1]. We started off with implementing a naive version of the algorithm mentioned in the paper. There are essentially 2 algorithms in the paper which differ in one of the phases.

The first algorithm which we will call $t - spanner$ works as follows:

Initially the spanner graph is empty with all vertices and no edges
Initially every vertex is its own cluster (clusters are sets of nodes)
Phase 1: Forming clusters

1. Sample clusters with a probability of $n^{-\frac{1}{k}}$. So now we have sampled clusters and non-sampled clusters.
2. Find nearest neighboring sampled cluster for each vertex
3. Adding edges to the spanner:
    a. If vertex is not adjacent to any sampled cluster:
        i. add the smallest edge from vertex to each of the neighboring non-sampled clusters to the spanner graph
        ii. delete all the other edges between this vertex and the non-sampled clusters
    b. If vertex is adjacent to any sampled cluster
        i. find the closest sampled cluster

            ii. let "closest_dist" be the weight to closest sampled cluster
            iii. add this edge to the spanner graph
            iv. delete all the other edges from this vertex to the closest sampled cluster
            v. For each of the neighboring non-sampled clusters,
                1. if the smallest edge is less than "closest_dist",
                    a. add the smallest edge to the spanner and
                    b. delete any other edges to the non-sampled clusters.
    4. Remove intra-cluster edges
    5. The sampled clusters become the new clusters for the next iteration
    Run Phase 1 k-1 times

Phase 2 : Vertex-cluster joining

- For each vertex, find the smallest edge from this vertex to all the clusters
- Add these edges to the spanner graph (also delete these edge from original graph)
- Remove all remaining edges incident on this vertex

The second algorithm (we'll call it "cluster-cluster") is almost the same as $t - spanner$ but is different in Phase 2.

In "cluster-cluster", we run Phase 1 $\left[\frac{k}{2}\right]$ times. And Phase 2 goes as follows:

Phase 2 : Cluster-cluster joining

- For each cluster, find the smallest edge connecting this cluster to all the other cluster
- Add these edges to the spanner graph (also delete these edges from the original graph)
- Remove all other edges which are incident on this cluster

In the paper, the authors have mentioned that this variation can save a factor of 2 in the number of edges in t-spanner but this variation does not give us any asymptotic improvement in the size.

The goal of phase 2 is to ensure that the clusters are connected.
$t - spanner$ achieves it by connecting every vertex in each cluster to every other cluster.
"cluster-cluster" as the name suggests adds only one edge between any 2 clusters. We essentially treat each cluster as a node and try to fully connect them.

2. Methodology

The plan is to:

1. **Implement a variation of the algorithm:**
    a. This could be a change of a data structure, slightly different parameters, etc.

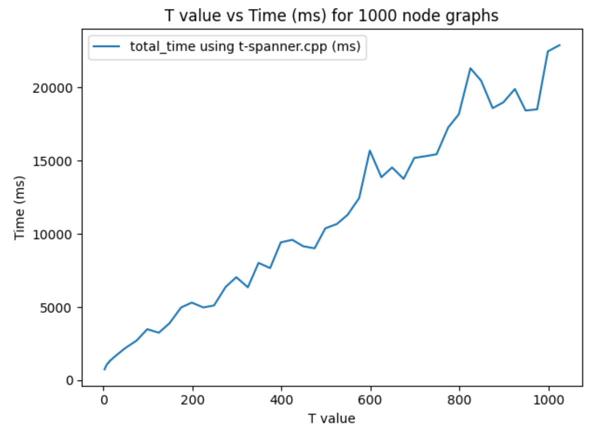
2. **Generate dense graphs:**
    a. Why dense graphs? Because the point of the t-spanner algo is to make subgraphs with number edges in the order of number of vertices. Sparse graphs already satisfy this property.
    b. How do we generate dense graphs? We randomly assign weights to the edges between any two vertices. The weights are taken from a normal distribution of mean 100 and variance 100. If any weight is <= 0, then the edge does not exist. The closer the mean is to 0 the sparser the graph gets.
    c. How do we test if a dense graph is a good candidate?
3. **Check Implementation for correctness:**
    a. We use Floyd-Warshall[2] to calculate the All Pairs Shortest Path in the original graph and the spanner graph. We define something called "spanner score" which is the maximum ratio between spanner distance and original distance between any two pairs of vertices.
    b. If the spanner score of the resultant spanner is less than or equal to the t value, then the algorithm is correct.
    c. We check for correctness on multiple dense graphs with varying t value and n value.
4. **Test against theoretical bounds:**
    a. The theoretical time complexity of the algorithm is $O(km)$.
    b. So the time should be linear in k => linear in t
    c. And the time should be linear in m => quadratic in n
    d. If we plot the time taken vs t or n, it should follow a shape similar to the expected one. This is another way to check the correctness of our implementations.
    e. Similarly we compare the expected number of edges vs the number of edges actually produced and see if they are similar in shape.
    f. The theoretical time complexity of the algorithm is $O(km)$ so the relation between t-value and total time it took to generate the t-spanner should be linear. We plot different t-values for a particular n value (1000) and check whether total time increases linearly as t-value increases.
5. **Comparing different variations:**
    a. We plot the times taken and spanner edges for each of the implementations against the dataset generated and compare them to see which one is the better implementation.
    b. Other parameters like phase wise time and edges were also plotted to further investigate the reason behind differences in time taken and difference in number of edges in spanner graph among the different implementations.
    c. Make hypothesis about the different implementations and test them out

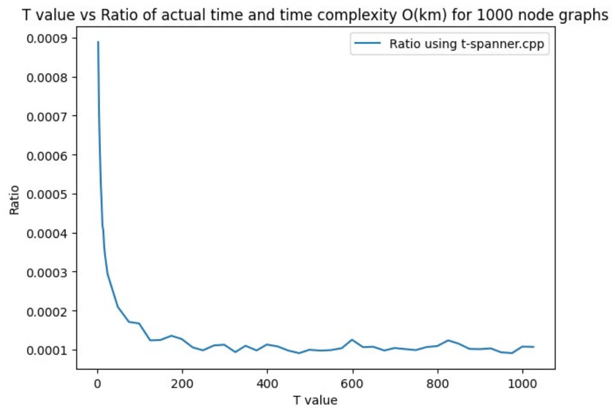
To facilitate this process, we created a test-suite which contains various commands which can be used to automatically generate test datasets, run the implementation on the dataset and store the output of the implementation in an accessible manner along with metrics such as time taken and number of spanner edges.

A great amount of time was put into making this test-suite as good as possible, this later came in handy as we wanted to test implementations with small variations with datasets of varying n values and t values.
Python scripts have been made to generate plots for t value vs any parameter and n value vs any parameter.

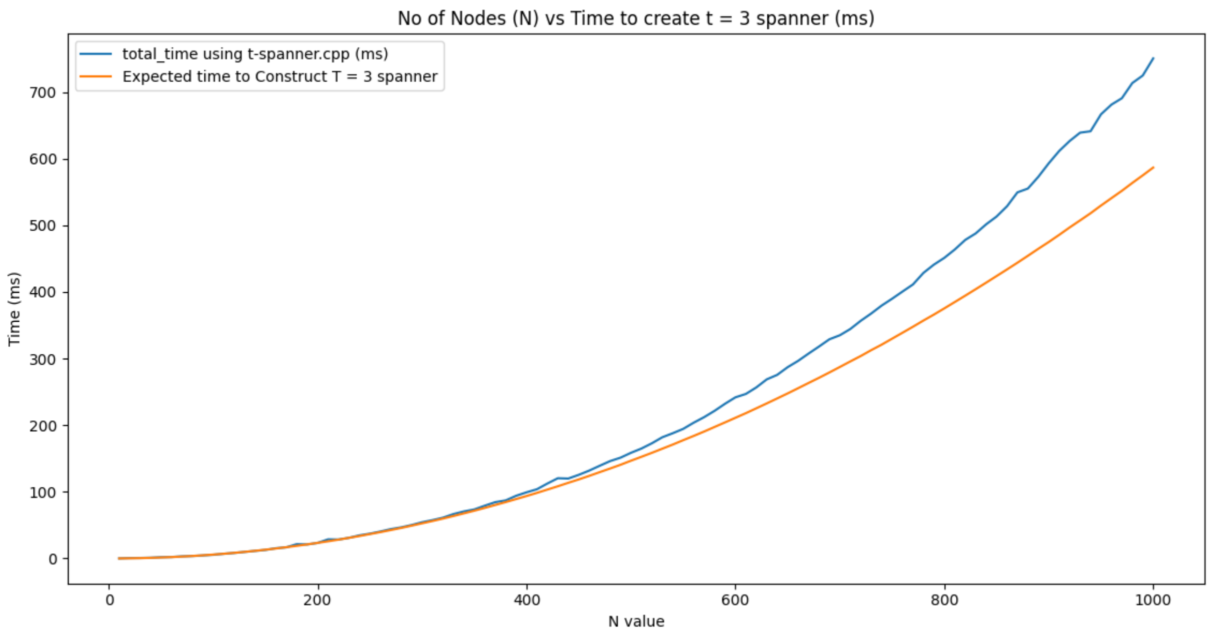
3. Experiments and Observations:

Test #1: The implementations should follow the theoretical trend
We wanted to test whether the implementations are following the theoretical bounds/trends and how far are they from the bounds in real life. The theoretical time complexity of the algorithm is $O(km)$ where $k = \frac{(t+1)}{2}$ and m is the number of edges in the original graph. As we can see that the time complexity is linear in k and therefore we can say that it is linear in t as well. Similarly, we can see that the complexity is also linear in m and if $m \sim O(n^2)$ which is true for dense graphs, then we can say that the time complexity should be quadratic in n for the same values of t. We test this by running our implementations on varying t-values for 1000 node dense graphs and then we average the amount of time it takes to calculate the t-spanner for a particular t-value.



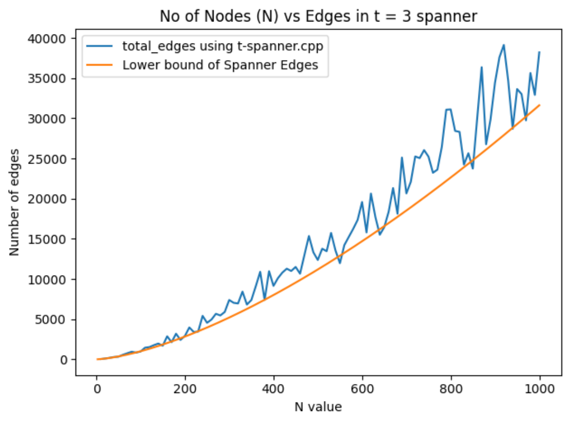


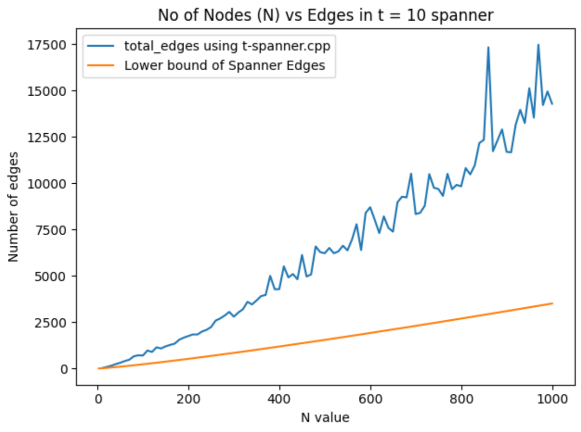
We can see that the graph (left) is roughly linear in the t-value and we can also verify this by plotting the ratio between the time it took to generate the spanner graph using t-spanner.cpp and the theoretical time complexity of the algorithm $O(km)$ where $k = \frac{(t+1)}{2}$. We can see in the ratio graph (right) that as t-value increases, the ratio becomes roughly constant. For smaller t-values, the ratio is not constant because the time to generate the spanner graph is dominated by other implementation factors and not the actual t-value.



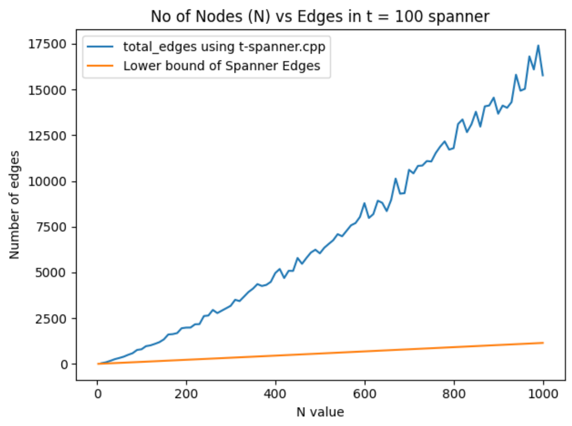
We can see that this graph is very close to the expected time which is quadratic in n and that verifies that our algorithm is $O(n^2)$ for dense graphs and constant t-values.

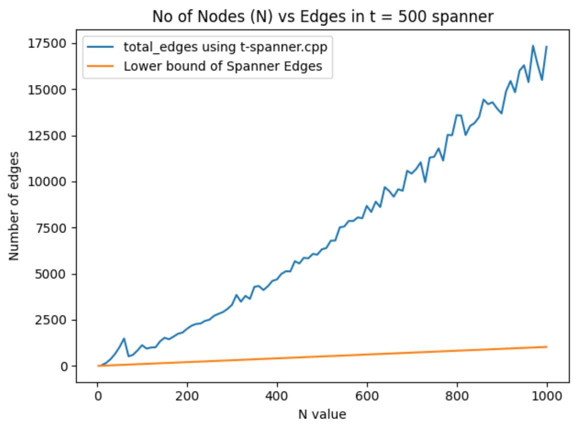
The number of edges in t-spanner is $O(k\, n^{1+\frac{1}{k}})$ so for bigger values of t, 1/k tends to 0 and therefore the number of edges becomes linear in n for big t-values. We can test whether our implementation follows this by running multiple tests, in each test we will choose a particular t value and then vary the n values. We will run these tests for t values of 3, 10, 100 and 500.





We can see that at t = 10, the graph resembles a linear line much more than at t = 3.

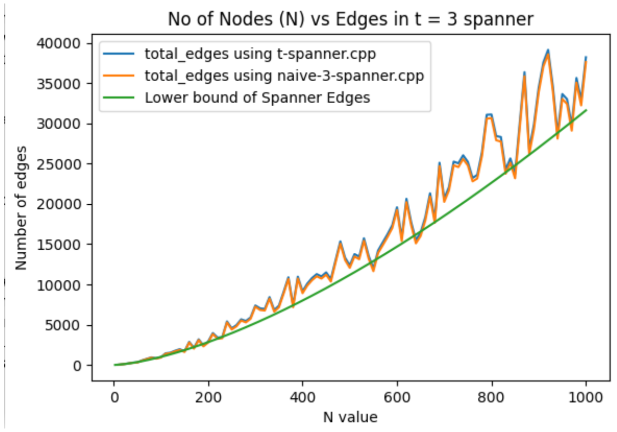


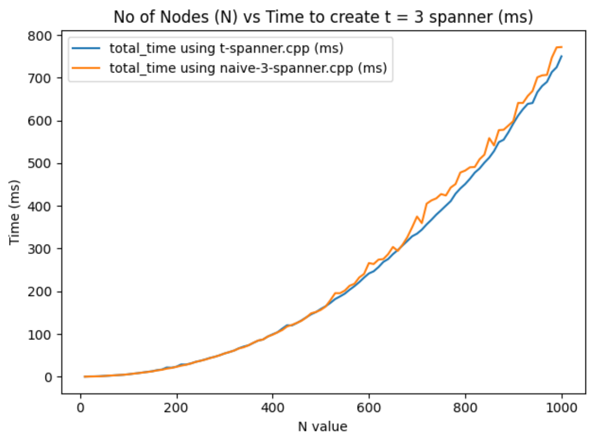


The linear relation becomes even more apparent as we plot the values for t = 100, 500 and see that the number of edges in the spanner graph does indeed increase linearly with n.

Test #2 : 3 spanner and t-spanner should be same at t = 3

We implemented the 3-spanner algorithm (which is simply the first iteration of $t - spanner$). Then we implemented the t-spanner algorithm. We compared the performance of the 2 algorithms when t = 3.

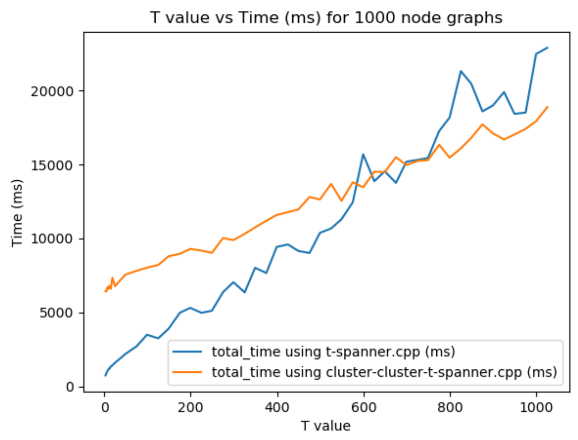


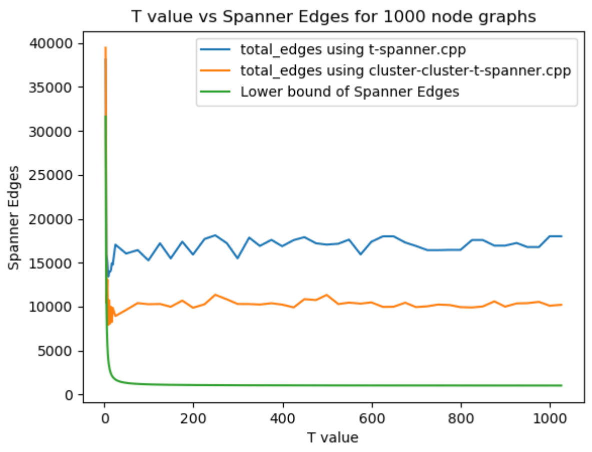


We can see that there isn't much of a difference between the time and number of spanner edges for both the implementations. This makes sense because for t = 3, t-spanner.cpp will just run one iteration in Phase 1 which is the same as that of 3-spanner.cpp. This further proves the correctness of the implementations as both of them are consistent with each other for t = 3.

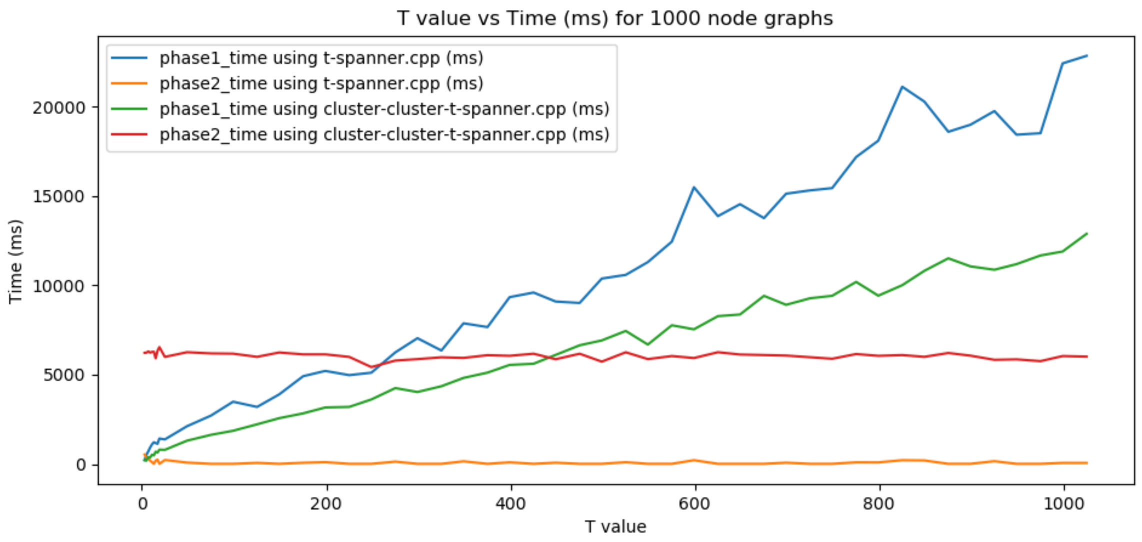
Test #3: Cluster-cluster Implementation

We implemented the second algorithm (the "cluster-cluster") and tested it out against the $t - spanner$. The paper predicts that asymptotically there should be no difference but practically it should give us half the number of edges. The following are the plots for time and spanner edges for increasing t value:





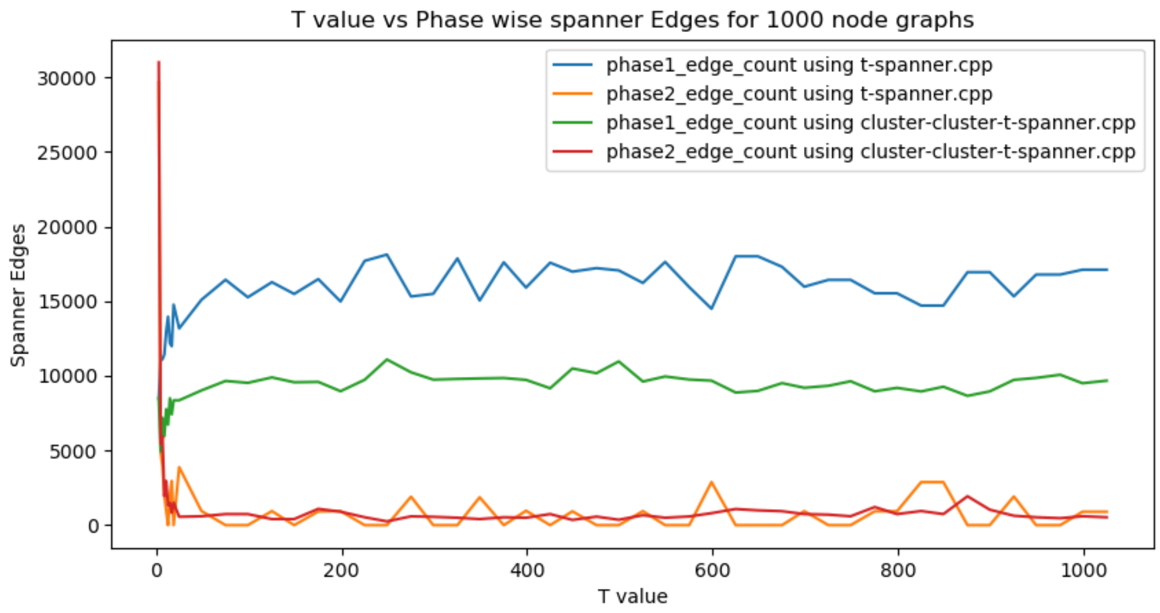
The time taken is initially high for "cluster-cluster" compared to $t - spanner$ but $t - spanner$ later overtakes "cluster-cluster". To investigate why this happened we plotted the phase wise time graph for the implementations. It's as follows:



We immediately notice that Phase 2 of "cluster-cluster" takes consistently longer time than $t - spanner$, however since we run the Phase-1 of "cluster-cluster" half as many times as $t - spanner$, it spends a lot less time in Phase-1 compared to $t - spanner$. The Phase-1 timings are ideally straight lines and if we fit a straight line to the plots we notice that the slope of Phase-1 of $t - spanner$ is around 2 times the slope of Phase-1 of "cluster-cluster".

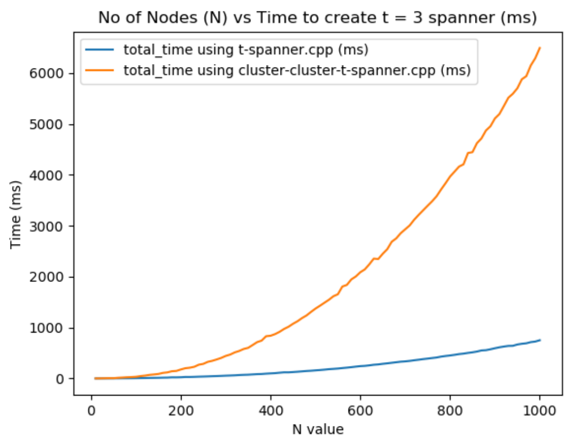
Initially Phase-2 time dominates therefore "cluster-cluster" seems slow but later when Phase-1 starts to dominate "cluster-cluster" becomes faster.

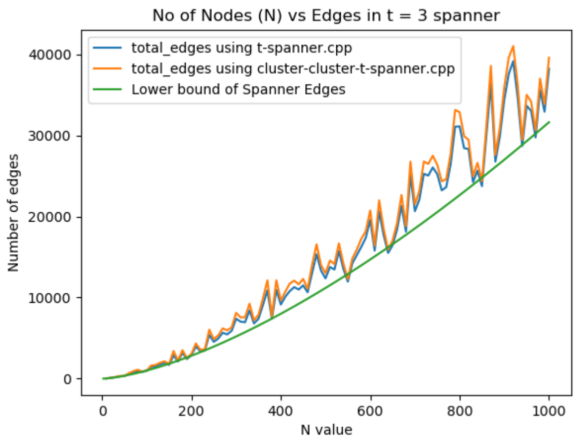
We notice that the spanner edges are roughly the same for small values of "t" like 3 to 5. But the difference is quickly apparent for larger values of "t". Where is the difference in edges coming from? We plotted the phase wise spanner edges to find out.

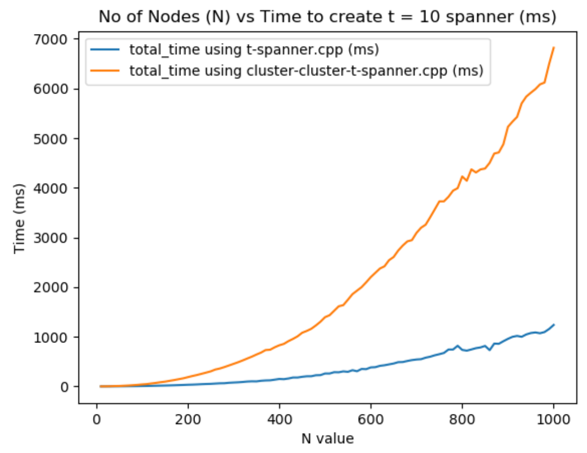


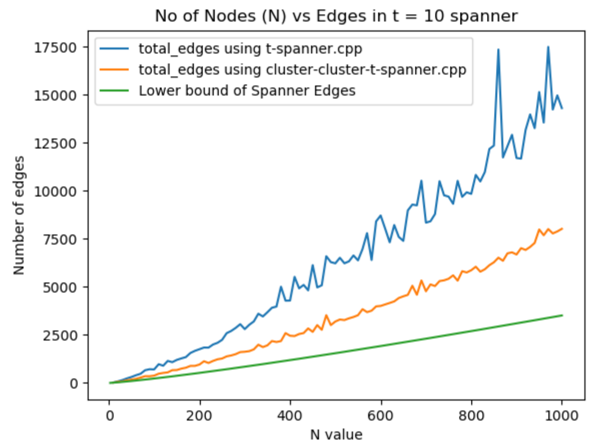
Given that the major algorithmic difference between $t - spanner$ and "cluster-cluster" is Phase-2, we notice that the number of edges added in Phase-2 are roughly the same. The major difference is in Phase-1 which we run only half as many times in "cluster-cluster" therefore we add only about half the edges that we would've added in $t - spanner$.

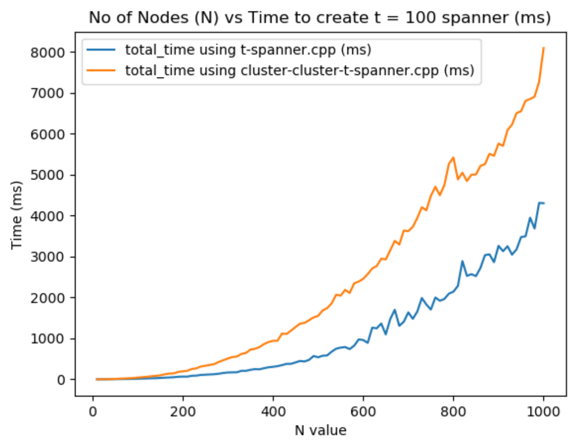
Then we plotted the time taken and edges vs increasing n values. We plotted this for t-values in [3, 10, 100, 500]. The graphs are as follows:

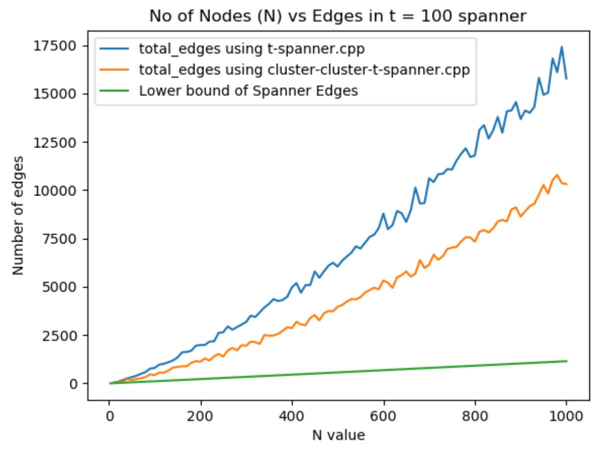


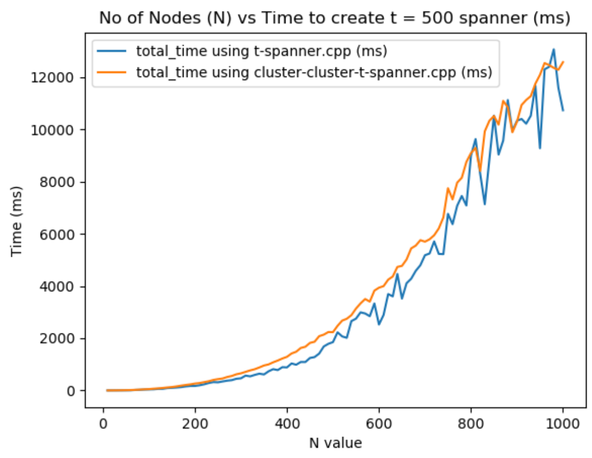


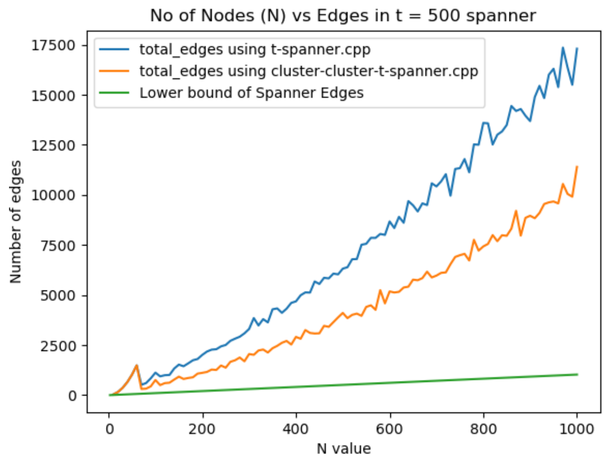










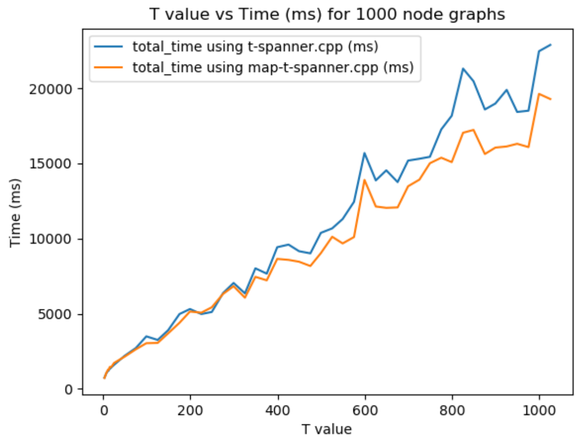


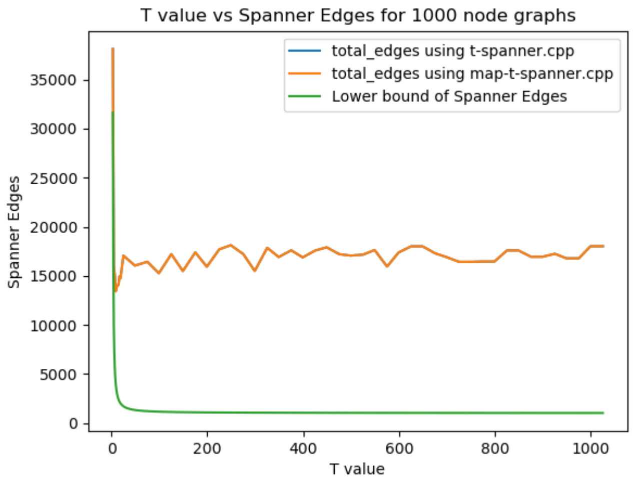
From the graphs we can observe that the spanner edges are initially the same at t = 3, but then start to differ significantly. We observe that the number of edges tends to be 1.5 or 2 times higher for $t - spanner$ compared to "cluster-cluster". This is due to the reasons explained previously.
We observe that the time taken by "cluster-cluster" is significantly higher at lower t-values, but catches up to $t - spanner$ for higher t-values. From the plots of t-value vs time, we notice that

"cluster-cluster" overtakes are around a t-value of 600, so for t-values higher than that "cluster-cluster" would be faster.
From this test we conclude that time taken by $t - spanner$ is better for smaller t-values and "cluster-cluster" for larger t-values. If reducing the number of edges is a priority then "cluster-cluster" is the preferred algorithm with significant difference, although it decreases the edges consistently for t-values larger than or around 10.

Test #4: Using STL map instead of STL set
A particular pattern which we seem to use a lot is finding and deleting edges that meet a certain condition from the adjacency list. The original implementation of the $t - spanner$ uses an array of sets (taken from c++ stl). Typically when implementing adjacency lists in c++ we use a vector of vectors or an array of vectors for the adjacency list, this is because we usually only traverse the adjacency list but here we needed to mutate the adjacency list.

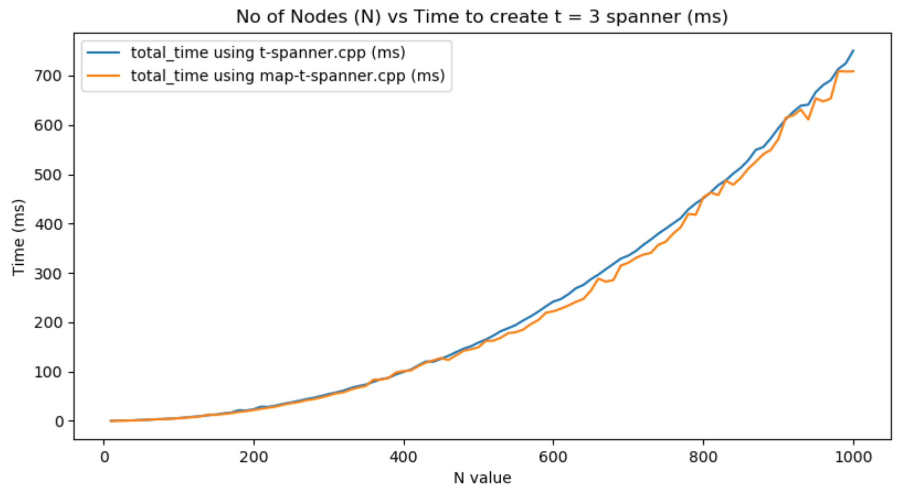
If the adjacency list was a vector of vectors (or array of vectors) it would take $O(n)$ to delete an edge from the adjacency list, and since we do this operation multiple times for each node, it would become $O(n^2)$ for one traversal of the adjacency list.
So in the basic implementation we used STL set, in order to achieve deletion in $O(log\ n)$ time. This would make the complexity in one iteration $O(nlog\ n)$.
We decided to see if using STL map can make a difference as its access time is $O(1)$ (in reality it is log(size of container). In hindsight we should've tested with unordered_map, but we didn't for reasons explained later).
Here are the results:

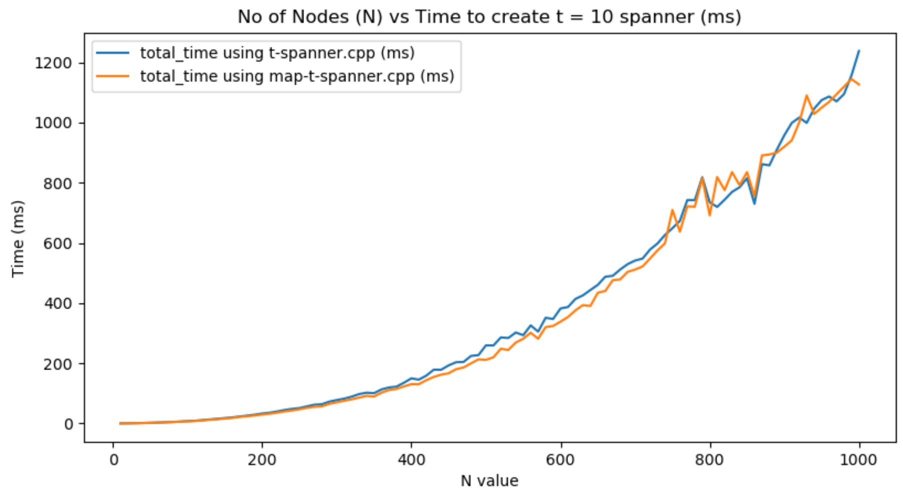


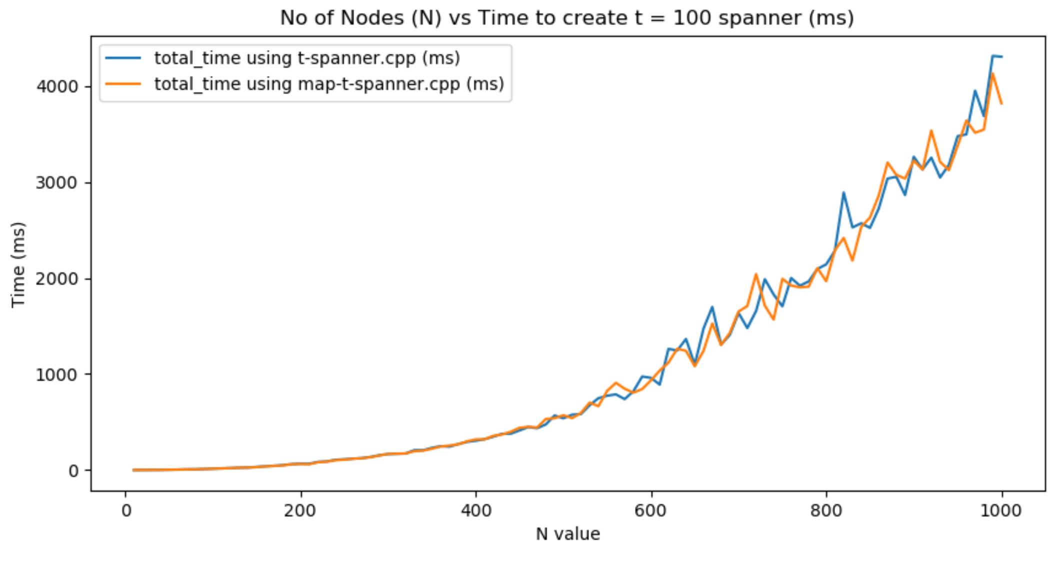


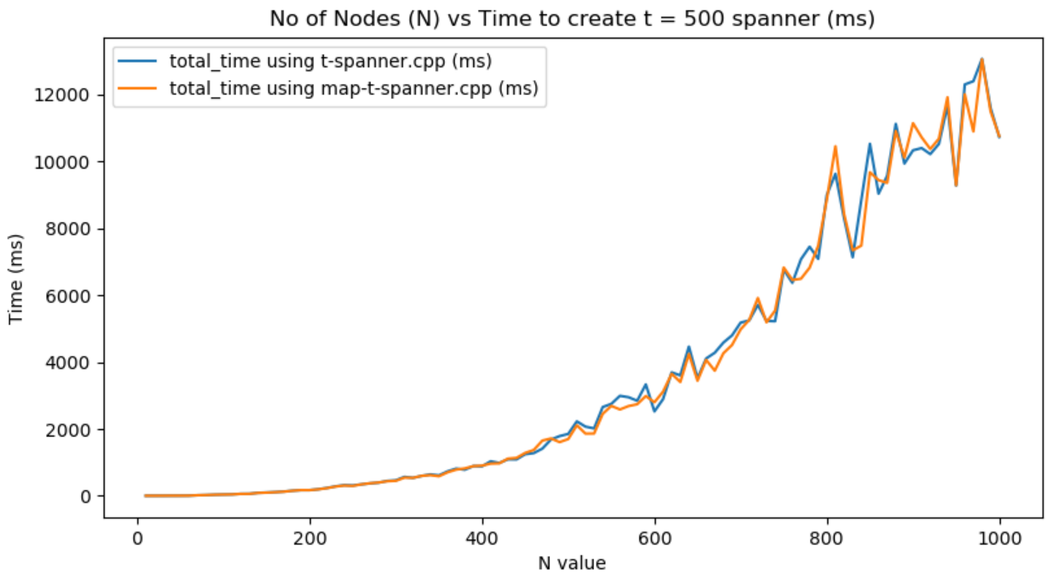
We see that the map implementation is faster for higher t-values. And as expected the number of edges are the same because there is no change to the actual algorithm.

We plotted for increasing n-values and the results are as follows:


No of Nodes (N) vs Time to create t = 3 spanner (ms)
total_time using t-spanner.cpp (ms)
total_time using map-t-spanner.cpp (ms)
Time (ms)
N value
0
100
200
300
400
500
600
700
0
200
400
600
800
1000


No of Nodes (N) vs Time to create t = 10 spanner (ms)
total_time using t-spanner.cpp (ms)
total_time using map-t-spanner.cpp (ms)
Time (ms)
N value
0
200
400
600
800
1000
1200
0
200
400
600
800
1000


No of Nodes (N) vs Time to create t = 100 spanner (ms)
total_time using t-spanner.cpp (ms)
total_time using map-t-spanner.cpp (ms)
Time (ms)
N value
0
1000
2000
3000
4000
0
200
400
600
800
1000

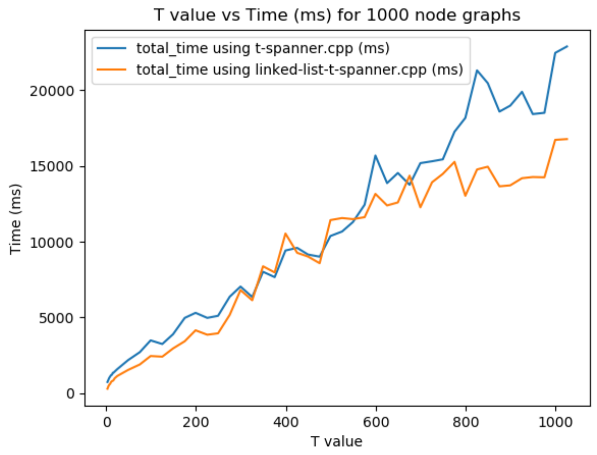


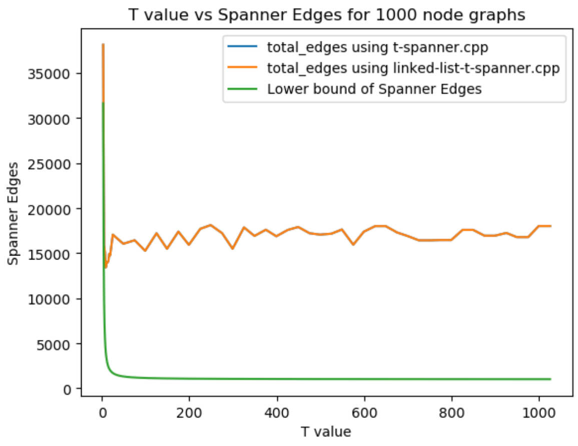
From the plots we can see that there isn't much of an improvement while using map instead of set. We planned on implementing it using unordered_map of STL, but we already implemented the linked list version of the adjacency list, whose results convinced us that we wouldn't need to implement the unordered_map version.

Test #5: Linked list implementation should be better than both map and set implementation

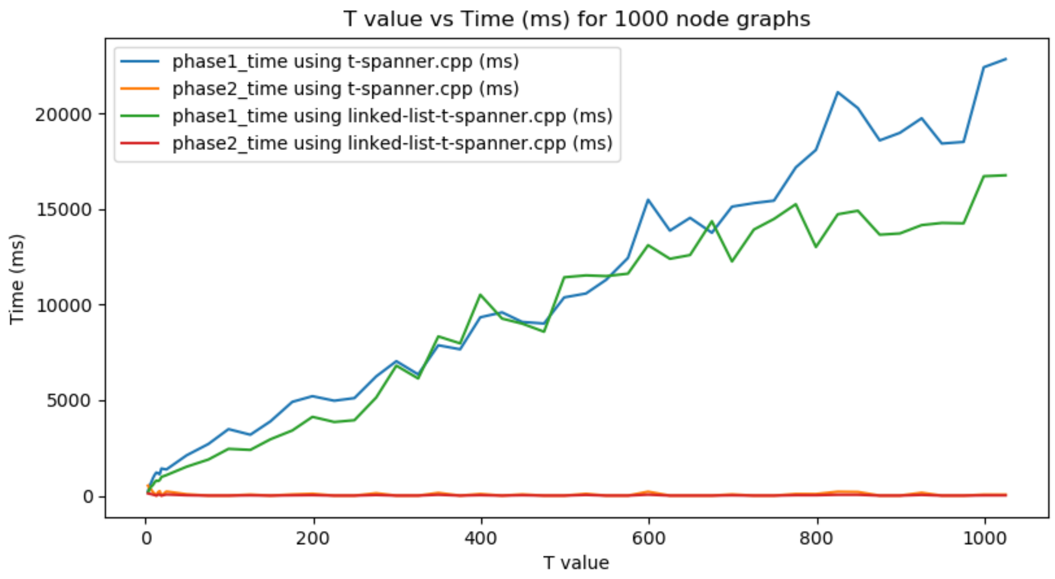
This data structure was suggested in the Sen Baswana paper. The data structure is an array of linked lists where every node is an edge and each edge also has a link to its counterpart in a different list. So edge $(u,\ v,\ w)$ has a pointer to the edge $(v,\ u,\ w)$. This is done because when we delete one edge from the adjacency list we only delete the forward edge but since we are dealing with undirected graphs we also need to delete the backward edge.
Deleting an edge from this adjacency list is $O(1)$ if we have access to the iterator (wrapper around a pointer) of that node, and since we are traversing the whole adjacency list we will readily have access to it when we are deleting the corresponding edge. We made the adjacency list as an array of STL lists. Each of these lists is a list of struct Node. Each of these Node structures contains the other vertex, the weight of the edge and the pointer (iterator) to the counterpart edge.

Comparing it against the set implementation gives the following results:



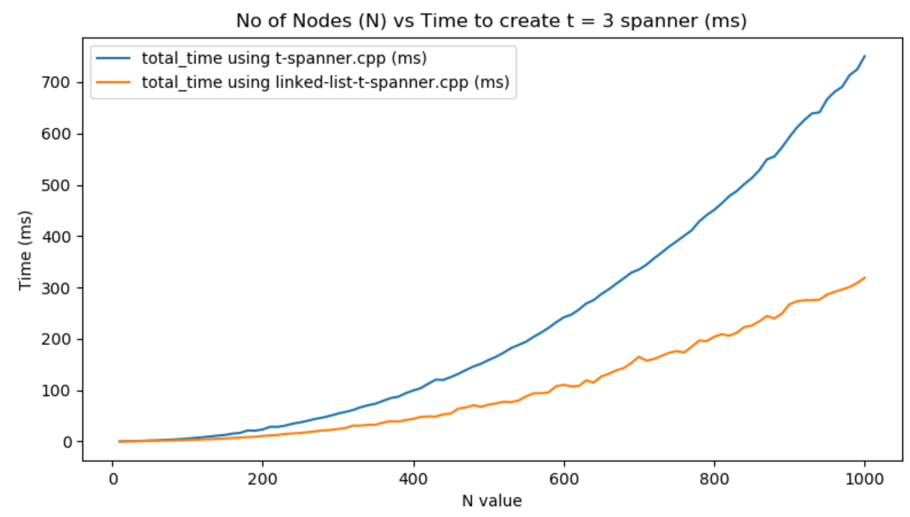


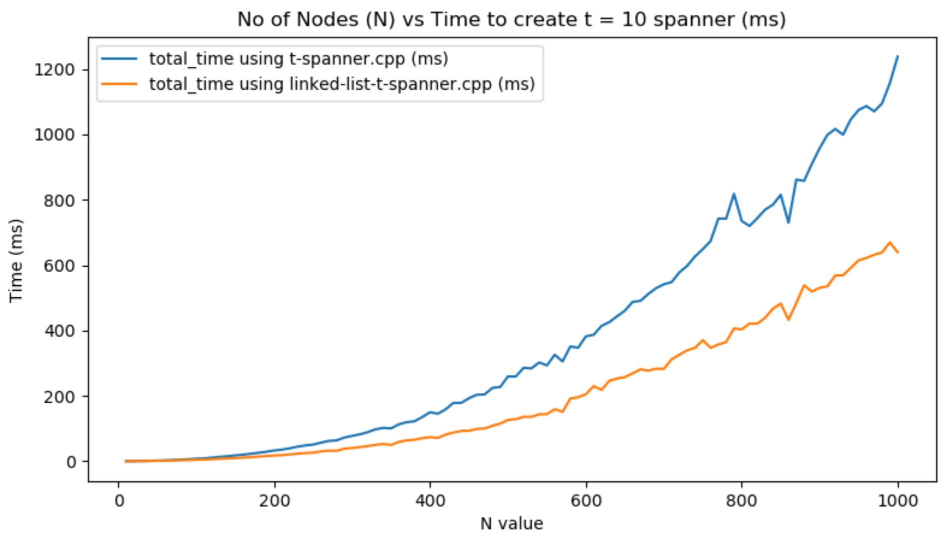
We observe that the number of edges stays the same as expected. However, the time taken is reduced significantly. For large t-values, there is a difference of multiple seconds. Where is the time improvement coming from?

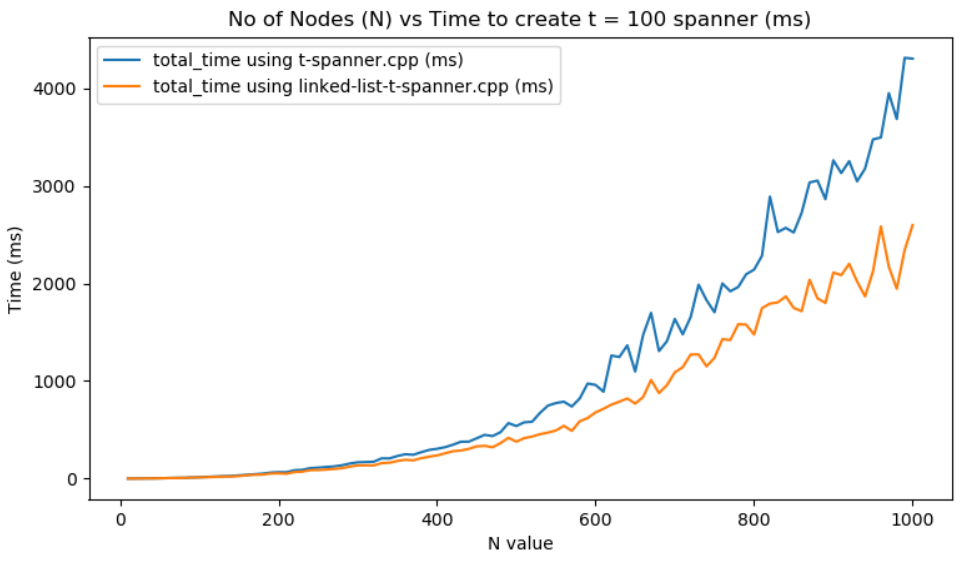


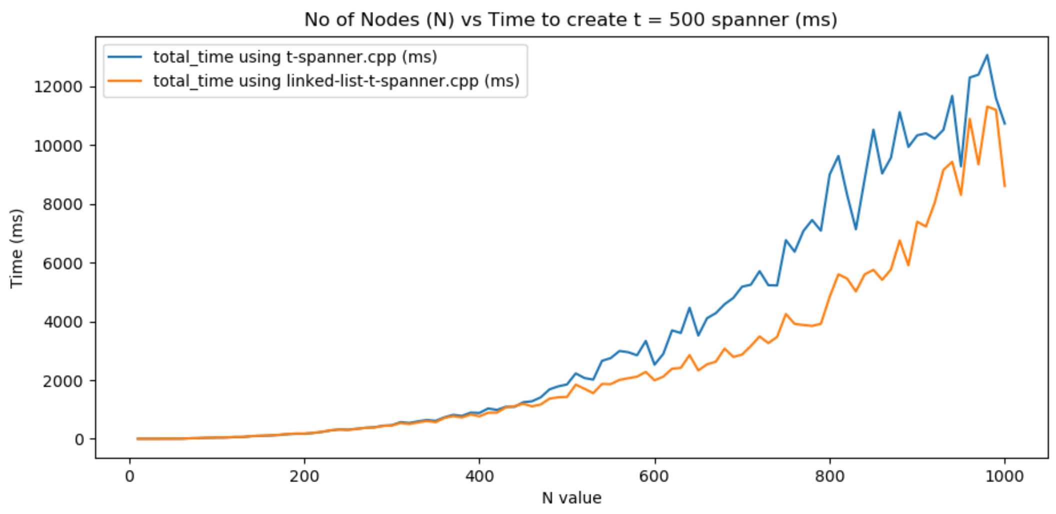
From the phase-wise plot we can say that the improvement is mostly from Phase-1. Phase-2 is almost the same. This is expected because most of the edge deletion is happening in Phase-1.

The following are plots with increasing n-value:

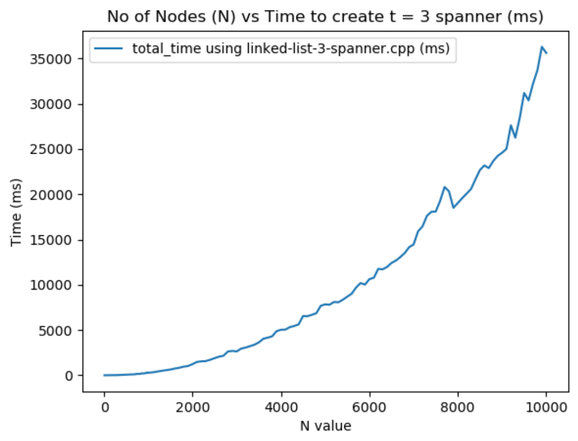

No of Nodes (N) vs Time to create t = 3 spanner (ms)
total_time using t-spanner.cpp (ms)
total_time using linked-list-t-spanner.cpp (ms)
Time (ms)
N value
0
100
200
300
400
500
600
700
0
200
400
600
800
1000

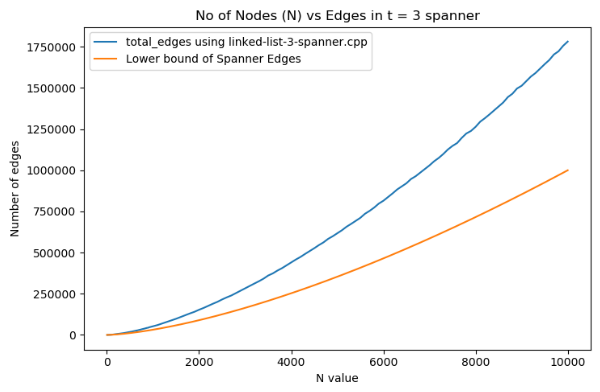

No of Nodes (N) vs Time to create t = 10 spanner (ms)
total_time using t-spanner.cpp (ms)
total_time using linked-list-t-spanner.cpp (ms)
Time (ms)
N value
0
200
400
600
800
1000
1200
0
200
400
600
800
1000


No of Nodes (N) vs Time to create t = 100 spanner (ms)
total_time using t-spanner.cpp (ms)
total_time using linked-list-t-spanner.cpp (ms)
Time (ms)
N value
0
1000
2000
3000
4000
0
200
400
600
800
1000

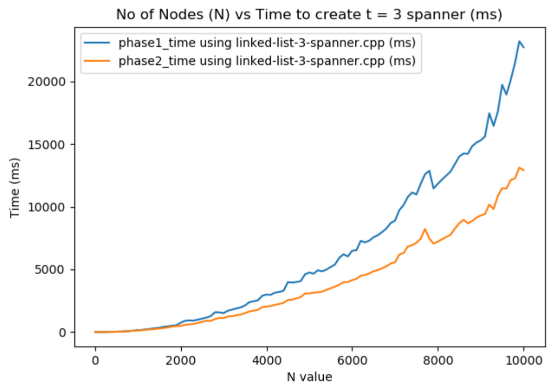


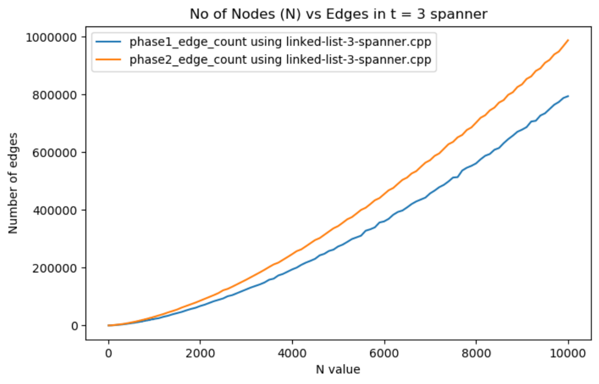
We see that linked-list implementation is consistently faster than set implementation. This is the implementation suggested by the paper and is also the fastest so far. So we dared to run it on the largest dataset we had (we actually ran the linked list version of 3-spanner on it, because that's what we felt would be safer.)
The result is as follows:





Running this took about a day. So we did not run any other implementations, so we don't really have anything else to compare other than the theoretical bounds. However here's the phase-wise breakdown:

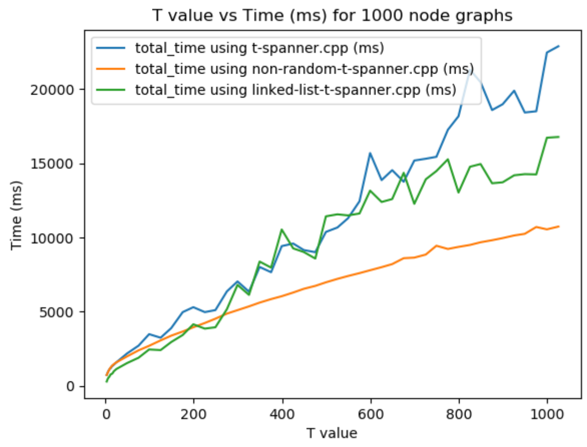


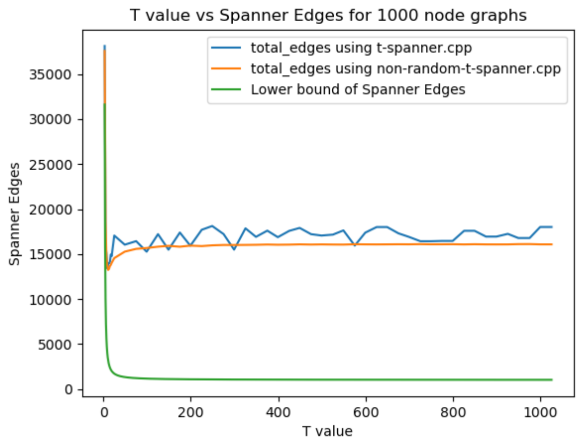


Test #6: Non-random should be faster
Now that we compared the two algorithms mentioned in the paper, we wanted to tweak these algorithms in a way that would improve either the time taken to run them or the number of edges in the spanner graph produced.

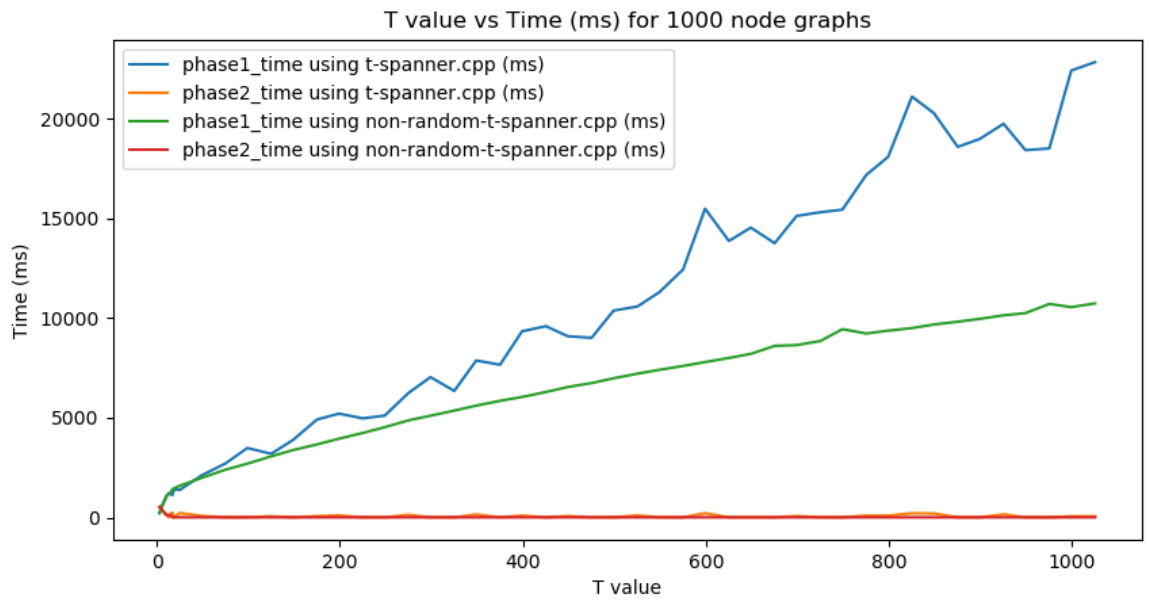
We noticed that we generate a random number for every cluster in each iteration of $t - spanner$. From experience we know that generating random is a somewhat slow process. So we wanted to change the implementation in such a way that it would not require the random numbers. The random number generation is used in sampling the clusters.
Instead we would deterministically select the first "S" clusters as the sampled clusters, where "S" is the expected number of clusters to be selected. The probability of selecting a cluster is $n^{-\frac{1}{k}}$. If there are C clusters then the expected number of sampled clusters would be $S = Cn^{-\frac{1}{k}}$.
Hypothesis: Picking deterministically the same number of clusters as the expected number of clusters achieved using the random method would be faster than the random method because we eliminate the need for random number generation.

So we ran the tests and the results are as follows:





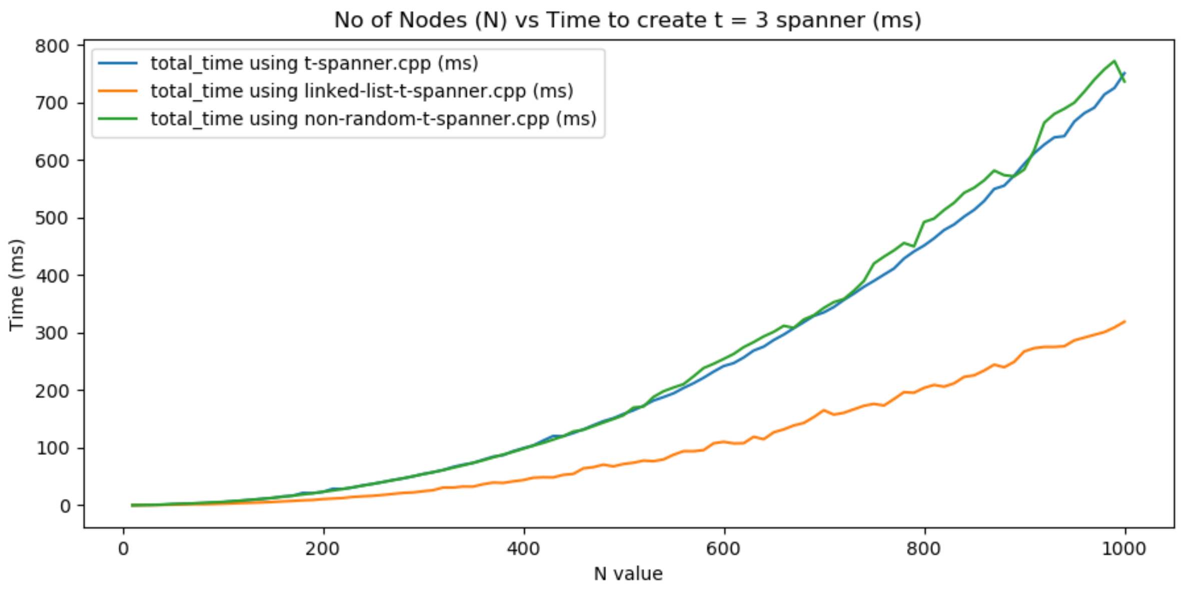
From the tests we see that for higher t-values, non-random-t-spanner performs significantly better than just $t - spanner$ and even linked-list-t-spanner which was the fastest implementation given in the paper that is not a parallel algorithm. The number of edges are roughly the same for the two.
Now, where is this improvement in time coming from? We plotted the phase-wise time graphs and here are the results:

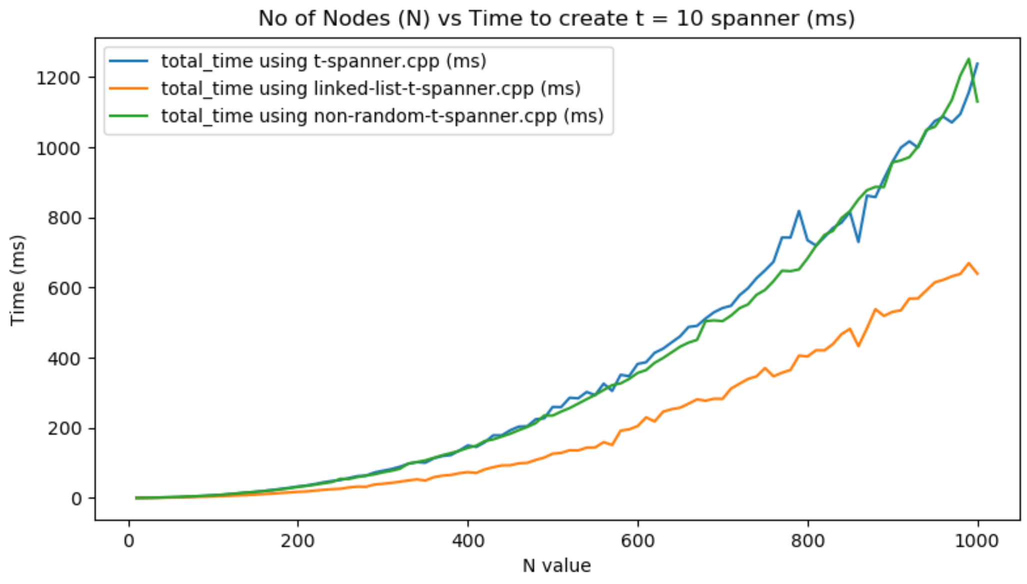


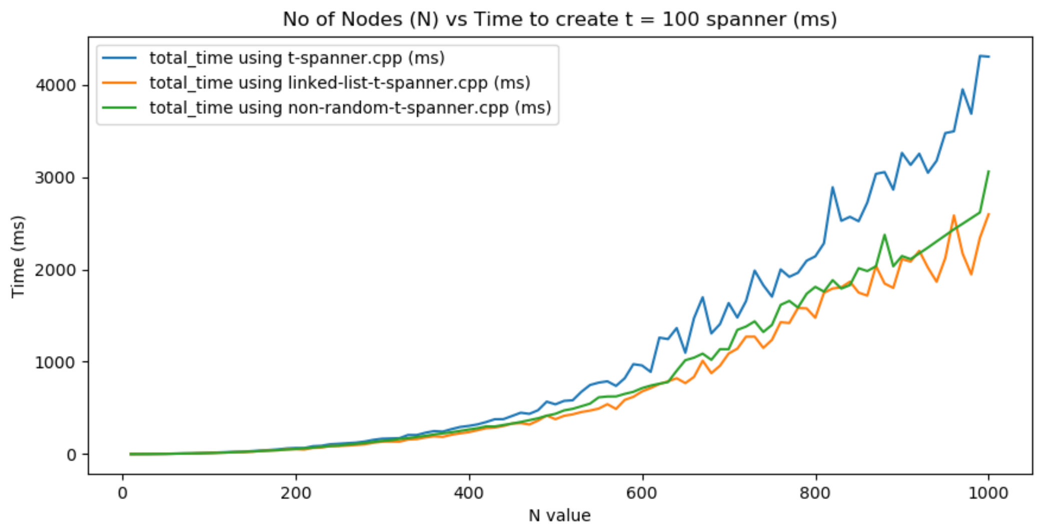
As we can see, Phase 2 for both the implementations takes the same amount of time, but there is a significant improvement in Phase 1 where the sampling happens. Therefore, from our observations we conclude our hypothesis is true. Not using the random function significantly improves the time.

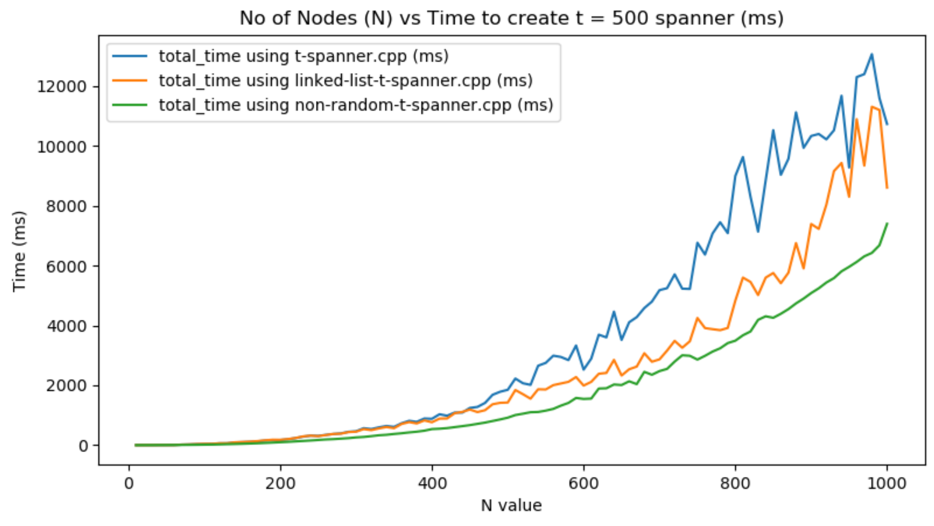
Although inconclusive, the number of edges also seem to be lower which is better.

Let's see how it performs against linked list implementation for increasing n values.




No of Nodes (N) vs Time to create t = 10 spanner (ms)
total_time using t-spanner.cpp (ms)
total_time using linked-list-t-spanner.cpp (ms)
total_time using non-random-t-spanner.cpp (ms)
Time (ms)
N value
0
200
400
600
800
1000
1200


No of Nodes (N) vs Time to create t = 100 spanner (ms)
total_time using t-spanner.cpp (ms)
total_time using linked-list-t-spanner.cpp (ms)
total_time using non-random-t-spanner.cpp (ms)
Time (ms)
N value
0
1000
2000
3000
4000


No of Nodes (N) vs Time to create t = 500 spanner (ms)
total_time using t-spanner.cpp (ms)
total_time using linked-list-t-spanner.cpp (ms)
total_time using non-random-t-spanner.cpp (ms)
Time (ms)
N value
0
2000
4000
6000
8000
10000
12000

From the above graphs we can see that as n value increases the difference in times increases significantly. For smaller values of t like 3 and 10. Linked list implementation is the best choice. But as we increase the t-value, the non-random implementation catches up to linked list implementation. At around t = 100, it is roughly the same. And at t = 500 we can see it is much better than linked list implementation.

The way we presented these plots is a bit misleading as linked list implementation and non-random strategy are not mutually exclusive. A linked list implementation of the non-random-t-spanner is expected to perform really well.

Conclusion:

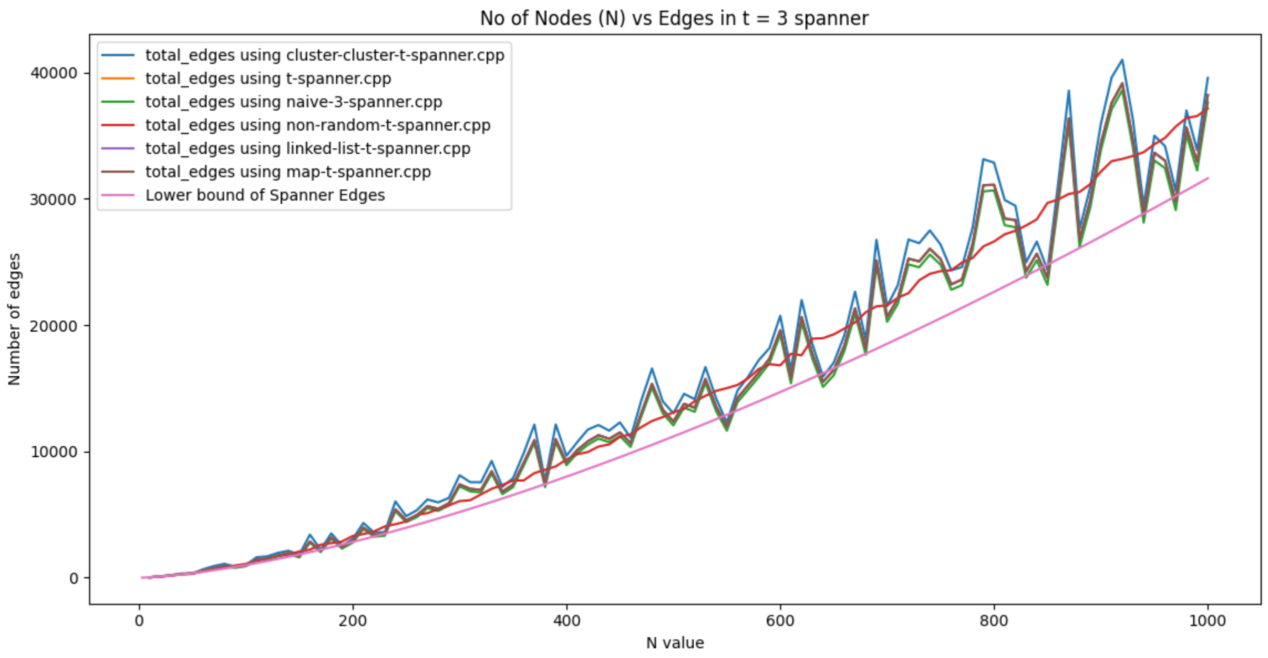
The base t-spanner algo with the linked list implementation is the fastest when t-value is small like 3 and gives the smallest number of edges as well. As we increase the t-value, non-random linked-list t-spanner should run the fastest and non-random-linked-list-cluster-cluster should give the smallest number of edges. But for much larger t-values, base cluster-cluster is faster than base t-spanner so the best implementation would become NON-RANDOM LINKED-LIST CLUSTER-CLUSTER [T-SPANNER]. This is faster than any non-parallel algorithm mentioned in the paper.

References:

[1] S. Baswana and S. Sen, A Simple and Linear Time Randomized Algorithm for Computing Sparse Spanners in Weighted Graphs, 3001 link

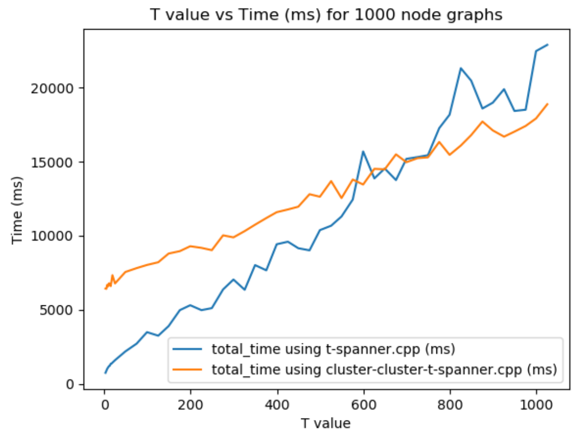
[2] Floyd-Warshall Algorithm link

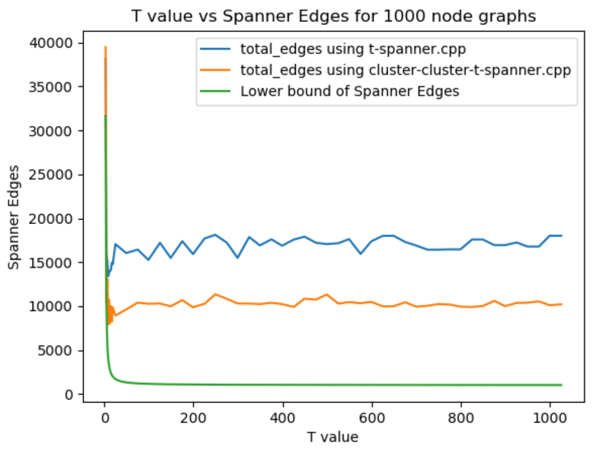
Graph with all the implementations



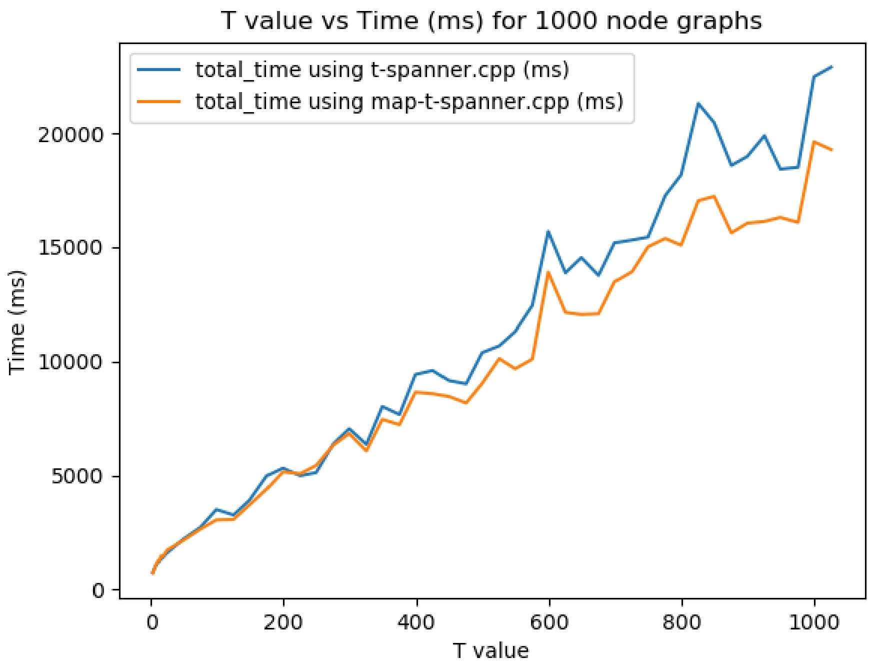
TLDR:

T-spanner (set) vs cluster-cluster:

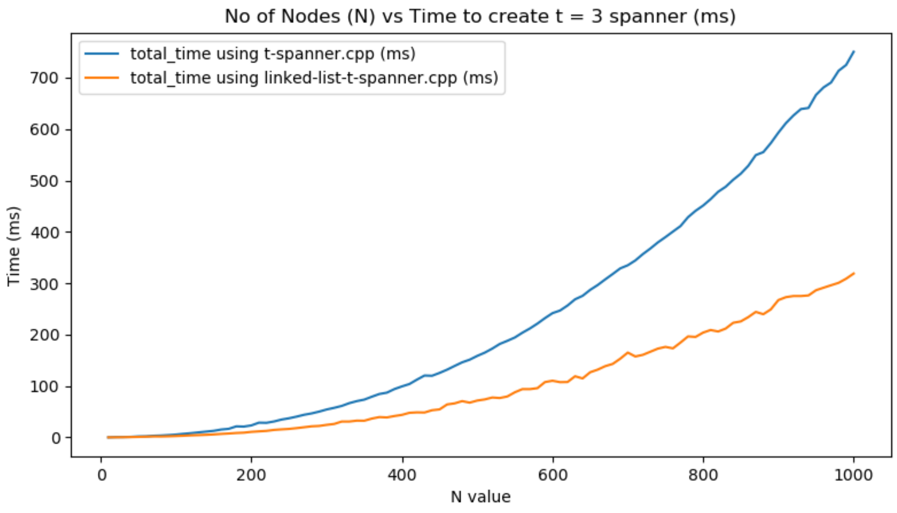




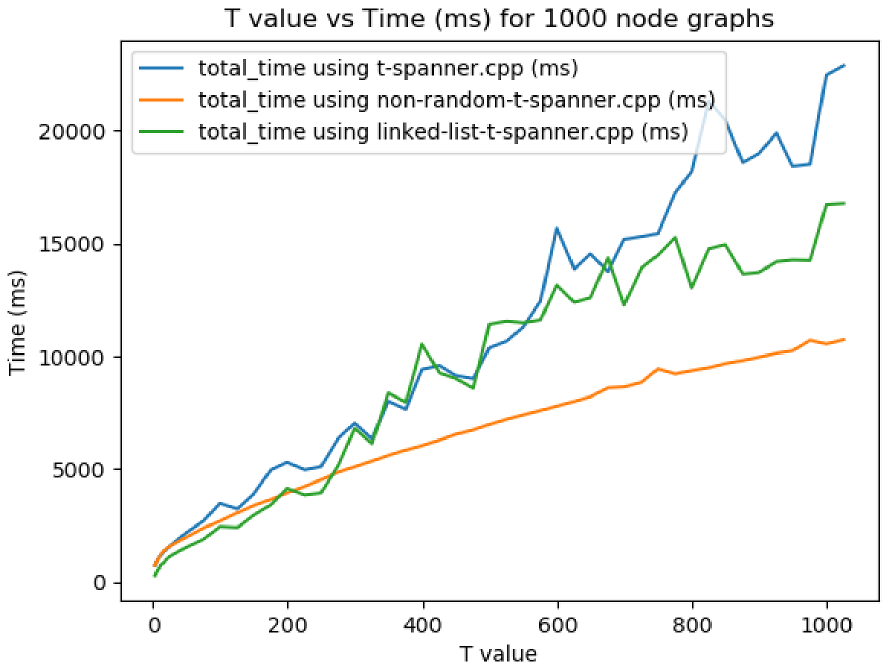
T-spanner (set) vs T-spanner (map):



T-spanner (set) vs T-spanner (linked list):



T-spanner (set) vs T-spanner (linked list) vs Non-random-t-spanner:


T value vs Time (ms) for 1000 node graphs
total_time using t-spanner.cpp (ms)
total_time using non-random-t-spanner.cpp (ms)
total_time using linked-list-t-spanner.cpp (ms)
Time (ms)
0
5000
10000
15000
20000
T value
0
200
400
600
800
1000