T-Spanner Construction

...

Vansh and Bhargav

T-Spanner Construction : 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.

The particular t-spanner algorithm on which our project is based on is the one by Baswana and Sen. We started off with implementing the naive version of the algorithm and then implemented different variations of it. The variations can be some difference in the data-structure used or it could be some change in the algorithm itself.

Why is a T-spanner needed?

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.

Sen and Baswana's Algorithm : Phase 1 - Cluster Formation

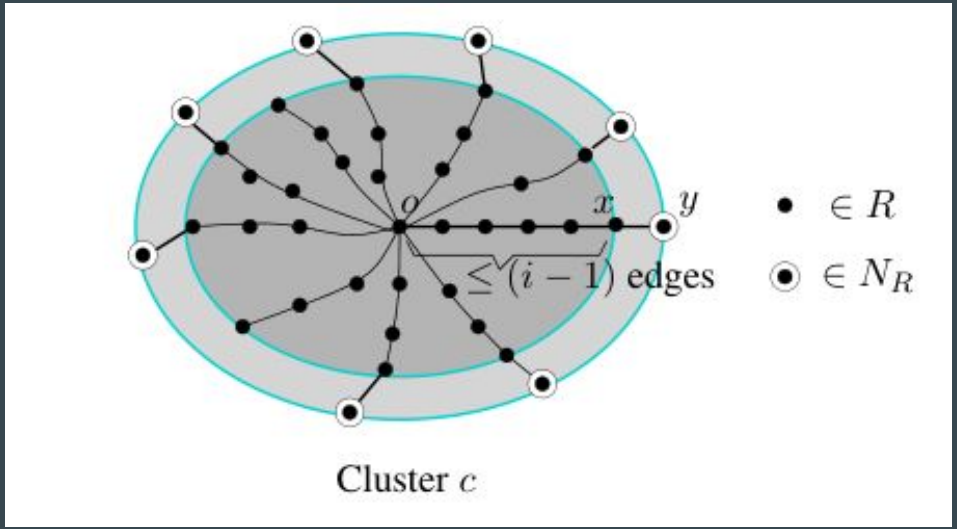
This phase executes k - 1 iterations. The ith begins with a set of clusters. Initially the spanner graph is empty and every vertex is it’s own cluster. Then we repeat these steps in each iteration:

- Sample clusters with a probability of n^(-1/k) So now we have sampled clusters and non-sampled clusters.
- Find nearest neighboring sampled cluster for each vertex
- Adding edges to the spanner
- Remove intra-cluster edges
- The sampled clusters become the new clusters for the next iteration

Sen and Baswana's Algorithm : 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

Below is an illustration of a cluster before the start of the ith iteration of Phase-1. Each cluster wil radius at most (i - 1) before the start of ith iteration



T-spanner Construction : Methodology - Basic Idea

We followed some simple steps as our methodology :

- Implement the original t-spanner algorithm by Baswana and Sen. Implement variations of these algorithms.
- Generate Dense Graphs
- Check Implementation for correctness
- Test against theoretical bounds
- Compare different variations

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 deal of time was spent in making the test-suite good to facilitate in testing hypothesis and comparing different implementations.
We also wrote some python scripts to automatically generate the plots from the output of the test-suite.

Methodology : Generating Dense Graphs

- 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.

- 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 with arbitrary mean and variance. If any weight is <= 0, then the edge does not exist. The closer the mean is to 0 the sparser the graph gets.

Methodology : Check Implementation for correctness

- We use Floyd-Warshall 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.

- If the spanner score of the resultant spanner is less than or equal to the t value, then the algorithm is correct.

- We check for correctness on multiple dense graphs with varying t value and n value.

Methodology : Comparing different variations

- 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.

- 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.

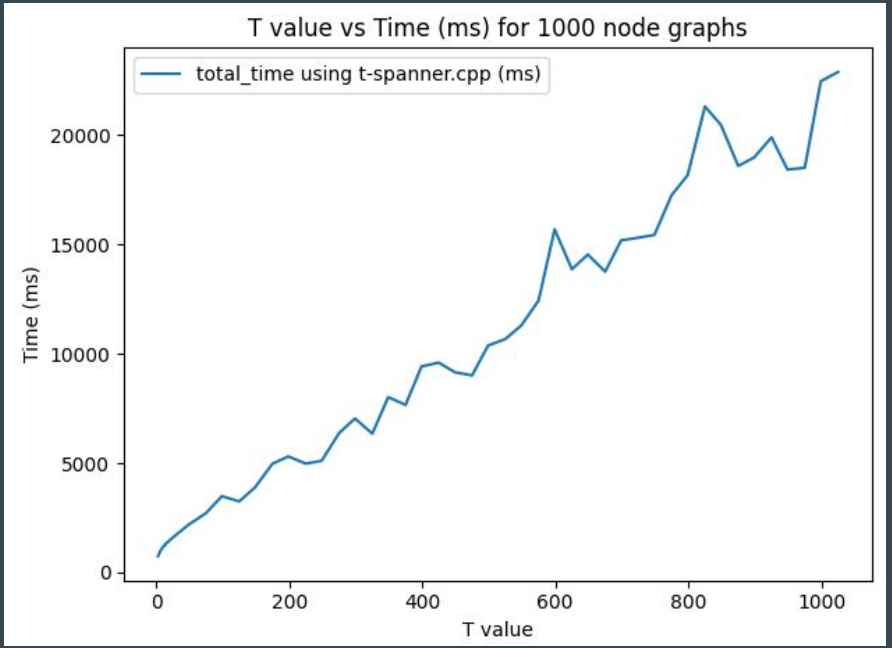
- Make hypothesis about the different implementations and test them out

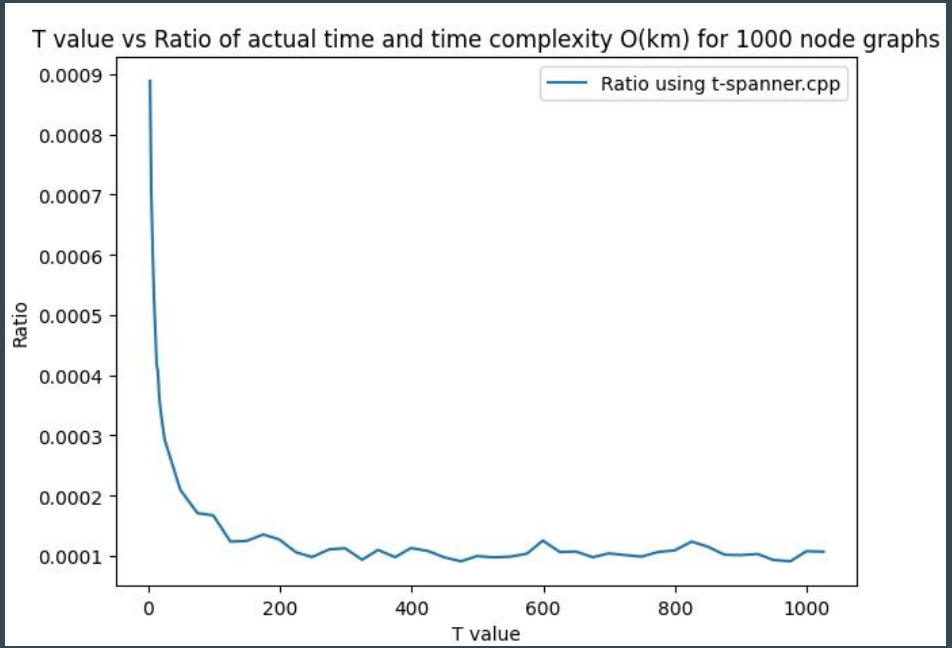
Test #1: Theoretical Trends - Some Pre-reqs

- Time Complexity of t-spanner : O(km) where k = (t + 1)/2 and m is the number of edges in the original Graph.
- This implies that the time complexity should be linear in t, i.e. - the graph between t-value and time taken to generate should the t-spanner should be a straight line.
- This also implies that for dense graphs where m ~ O(n^2). The relation between number of nodes and time taken should be quadratic.

- The number of edges in Spanner graph: O(kn^(1 + 1/k))
- This implies that for large values of t, the relation between n and number of edges in spanner graph should be approximately linear.

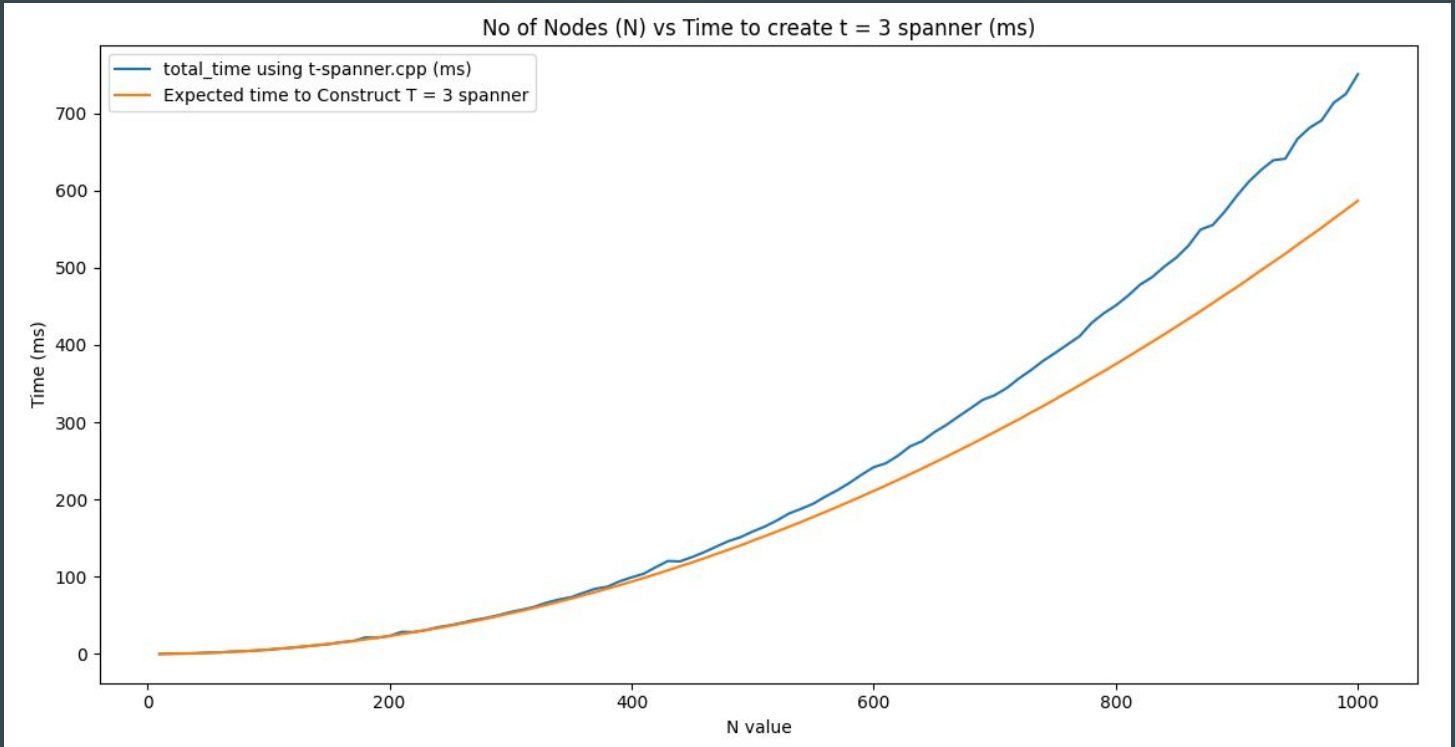
Test #1 : T-value vs Time Taken





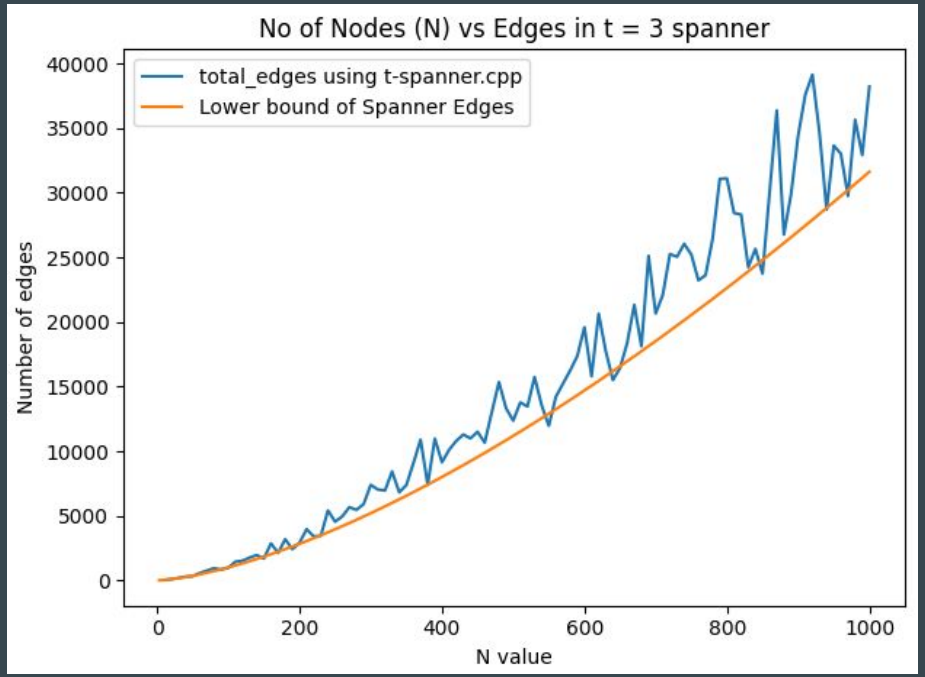
Test #1 : N-value vs Expected Time

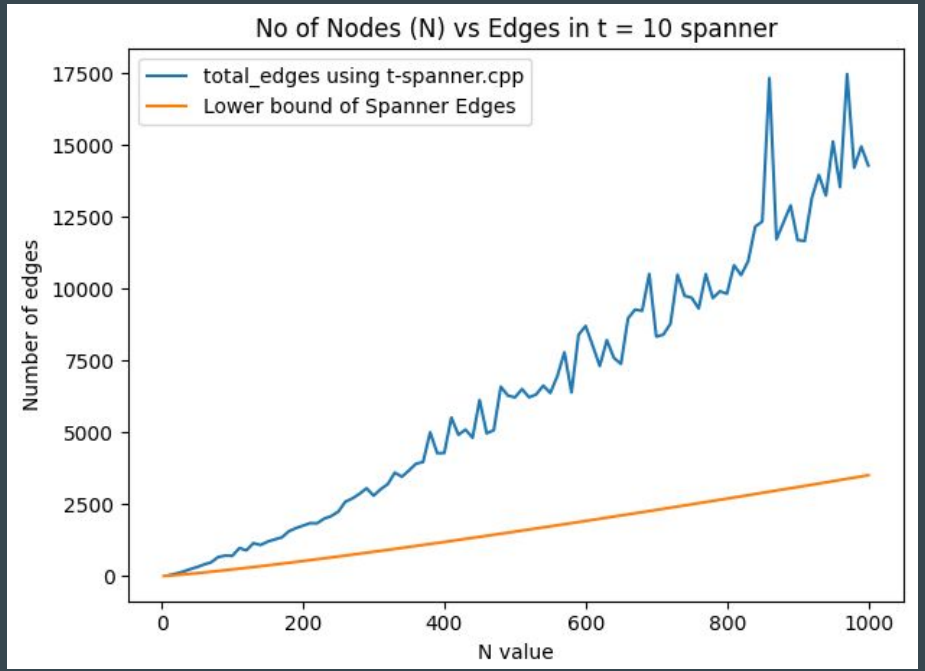
Here, the orange line is plotting the time complexity which is O(km). We can see from this graph that the



Test #1 : N-value vs Number of Spanner Edges (t = 3, 10)

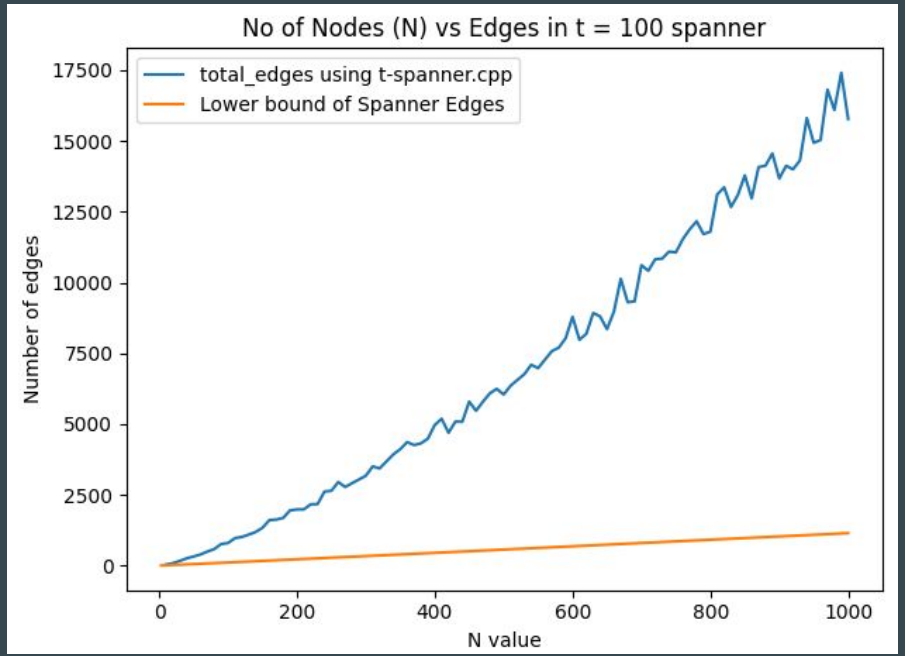
We can see that as we go from t = 3 to t = 10, the graph looks more like a line

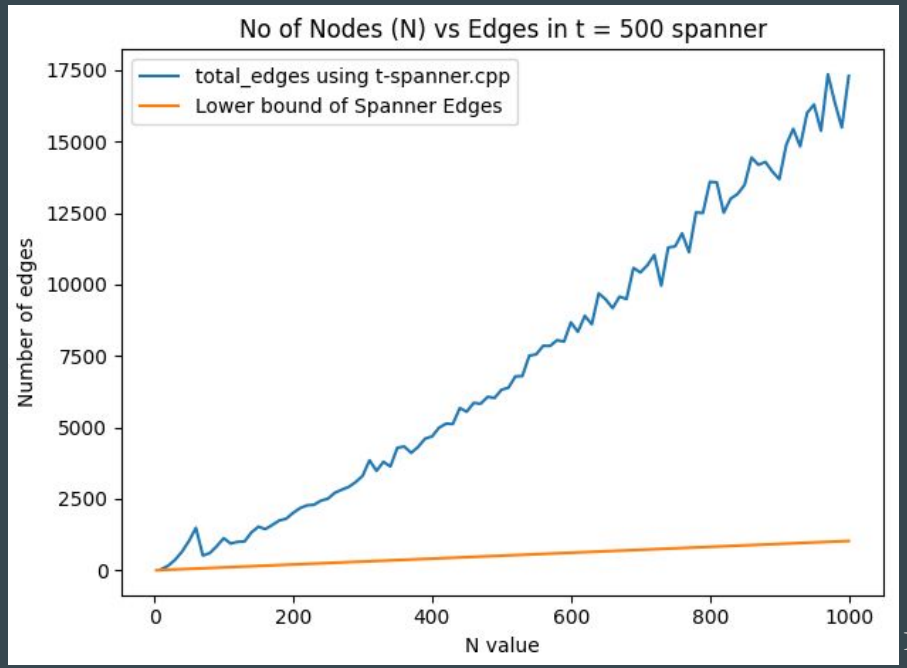




Test #1 : N-value vs Number of Spanner Edges (t = 100, 500)

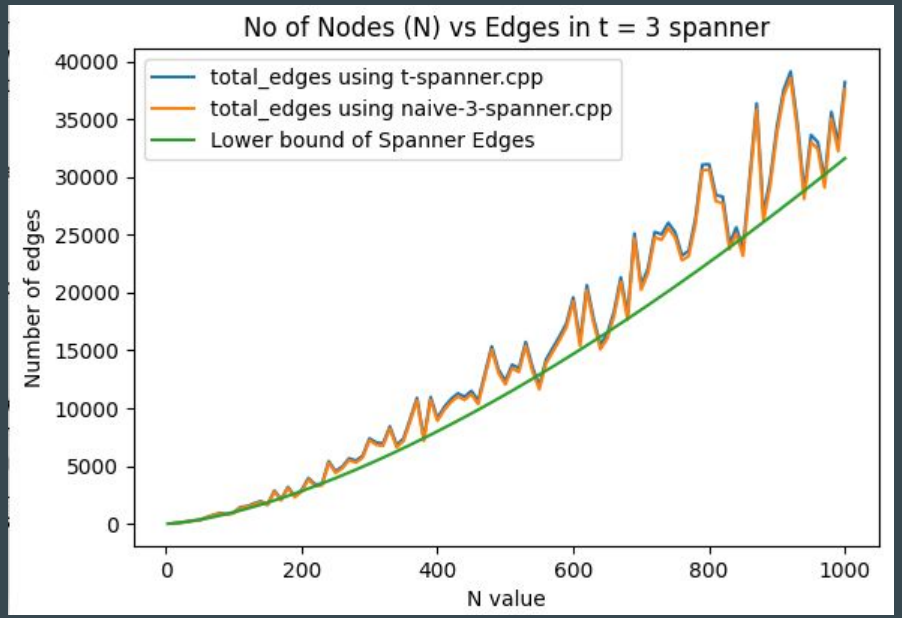
The difference becomes even more clear as go from smaller values of t like 3, 10 to bigger values like 100 or 500. At t = 500, we can easily see that the graph is almost a straight line.

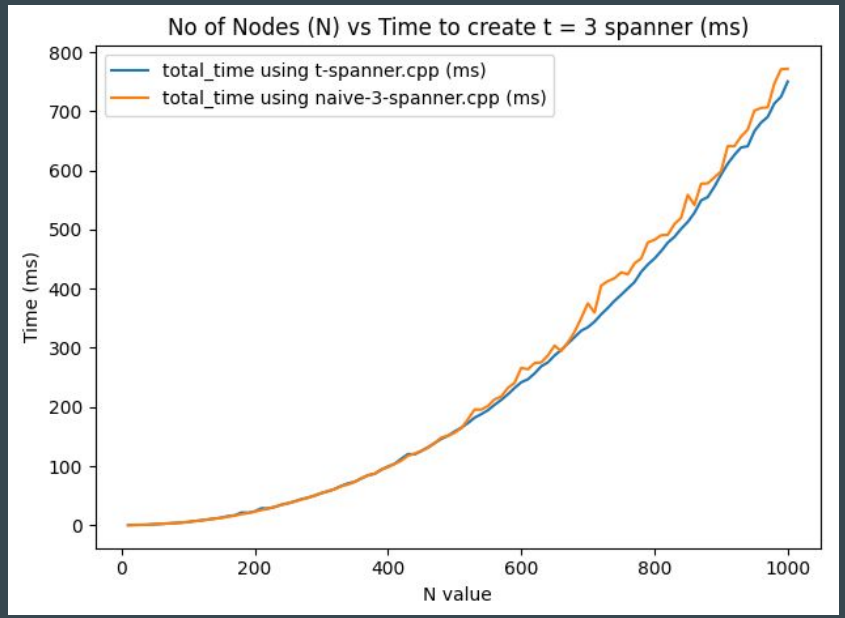




Test #2: 3-spanner vs t-spanner

We wanted to test whether 3-spanner and t-spanner at t = 3 would give the same performance for dense graphs or not. As we can see, from the graphs below that they do indeed give the similar performance both in terms of the number of edges and time taken.





Cluster-Cluster - description of new algorithm

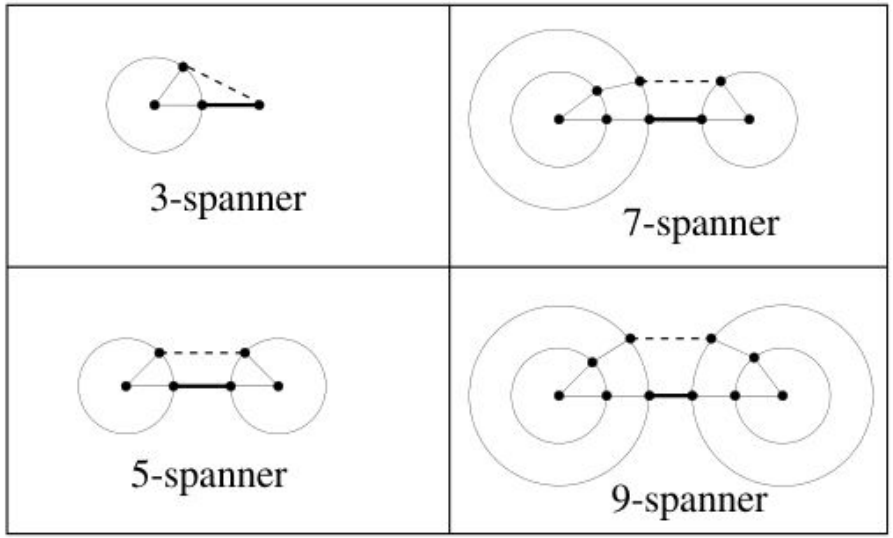
- In the original paper of the t-spanner algorithm, the authors have described another variation of the algorithm (we shall refer to this new algorithm by “cluster-cluster”).

- In “cluster-cluster”, we only run Phase-1 half the times and in Phase 2, instead of connecting every vertex to every cluster. We try and connect every cluster to all the other clusters (The authors call this phase Cluster-Cluster joining).

- For each pair of clusters, we find the smallest edge between these two clusters and delete all the other edges in between these clusters.

- The authors mention that while there is no asymptotic difference in the size of the spanner graph but we do save a factor of 2 in constructing the t-spanner when using “cluster-cluster”.

Cluster-Cluster : illustration



Edge already in the spanner

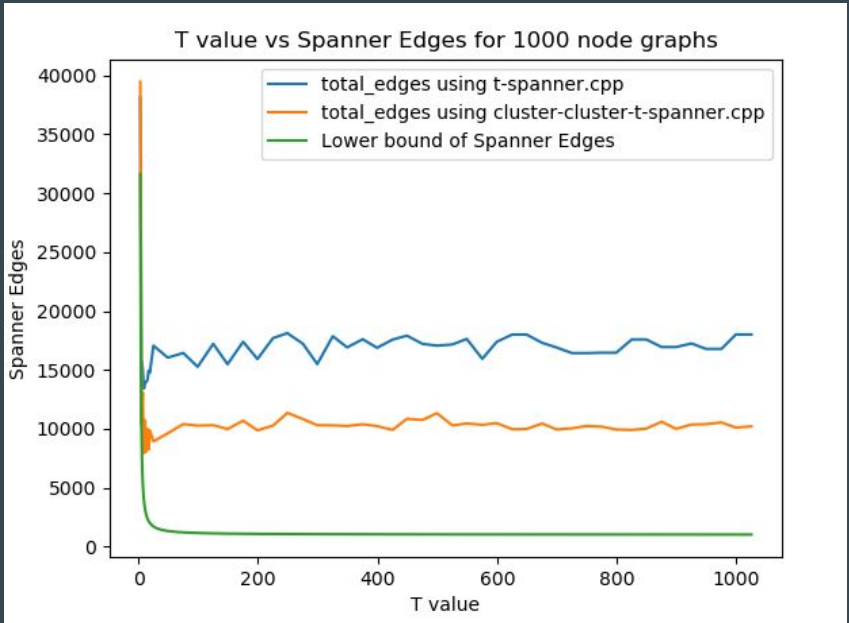
Edge added during *'cluster-cluster joining'*

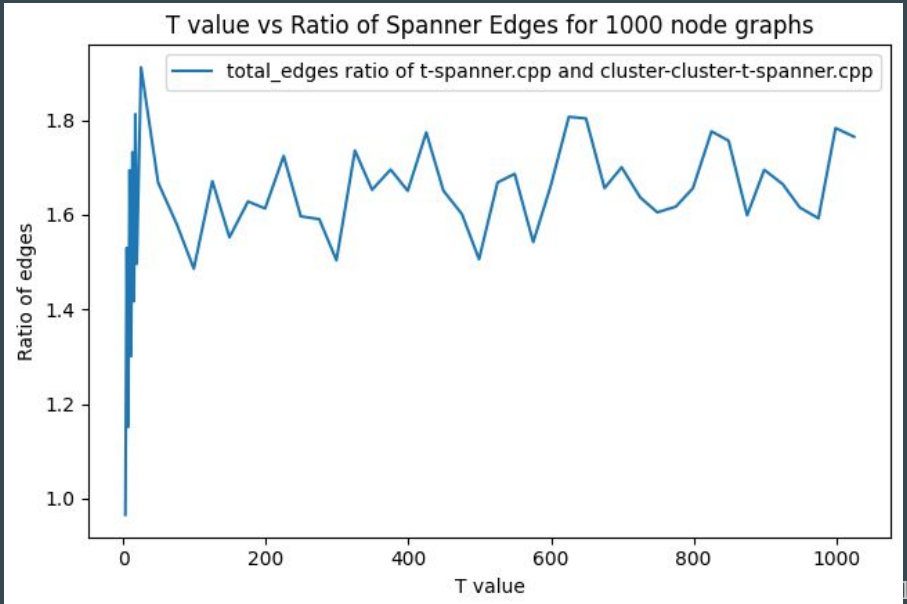
Edge not in the spanner

Figure 6: *'cluster-cluster joining'* phase of the new algorithm

Test #3: Cluster-Cluster vs T-spanner - spanner edges

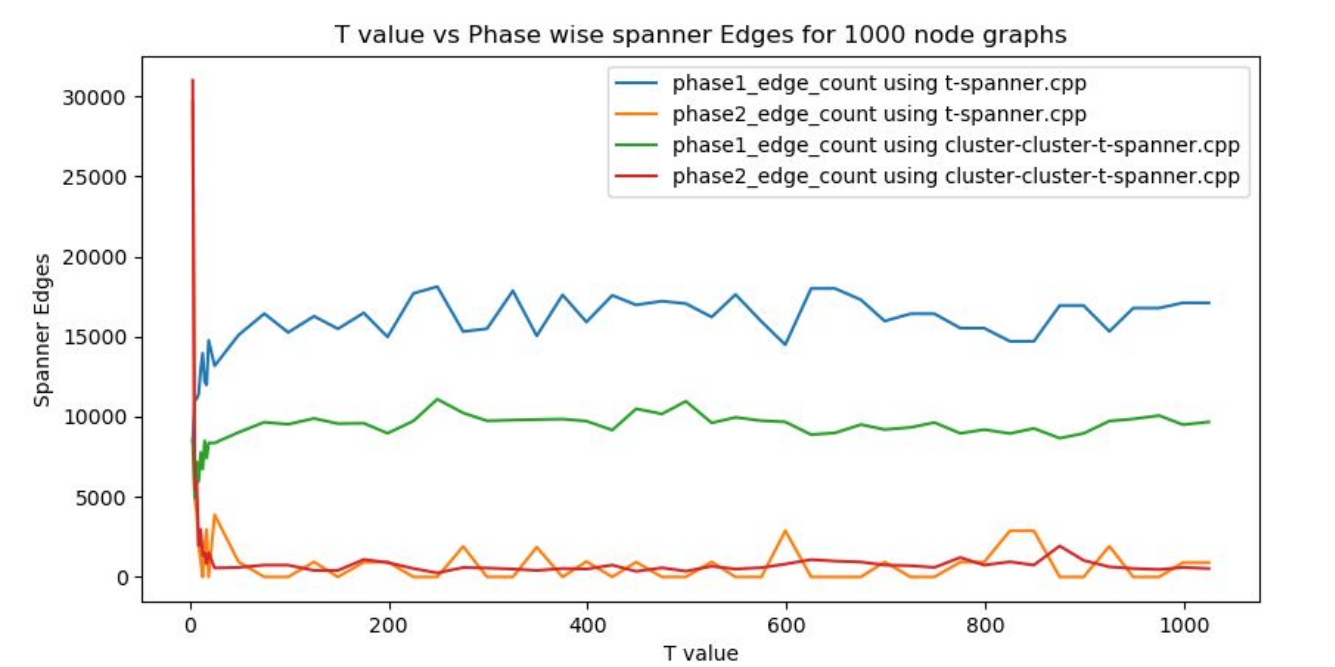
We wanted to test the claims of the authors and check whether we actually save a factor of 2 in the number of spanner edges when using cluster-cluster as opposed to when using the original t-spanner. We first plotted t-value vs total edges for a 1000 node graph. We found that the number of edges using cluster-cluster is indeed less but we wanted to investigate further...





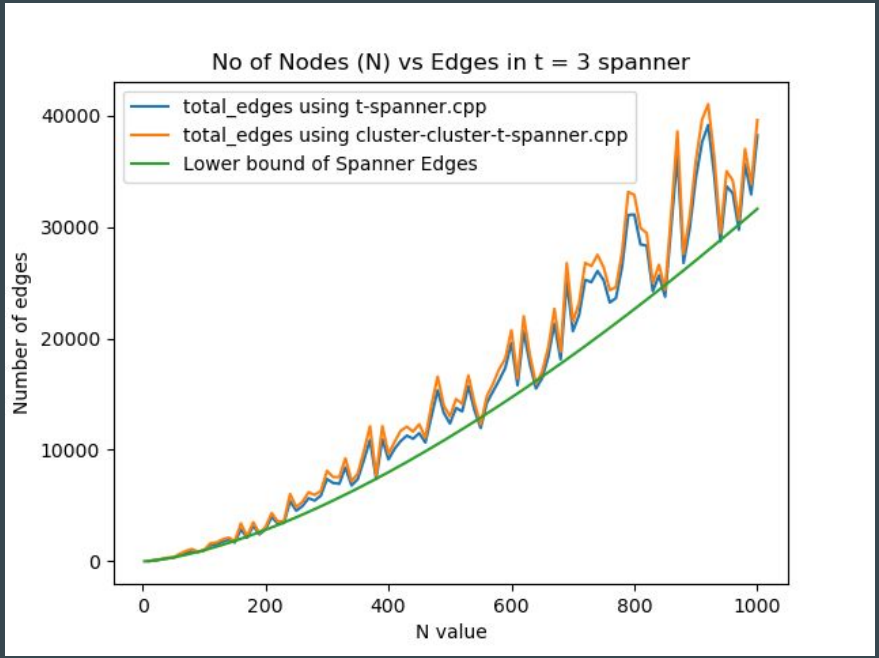
Test #3: Phase-wise edges

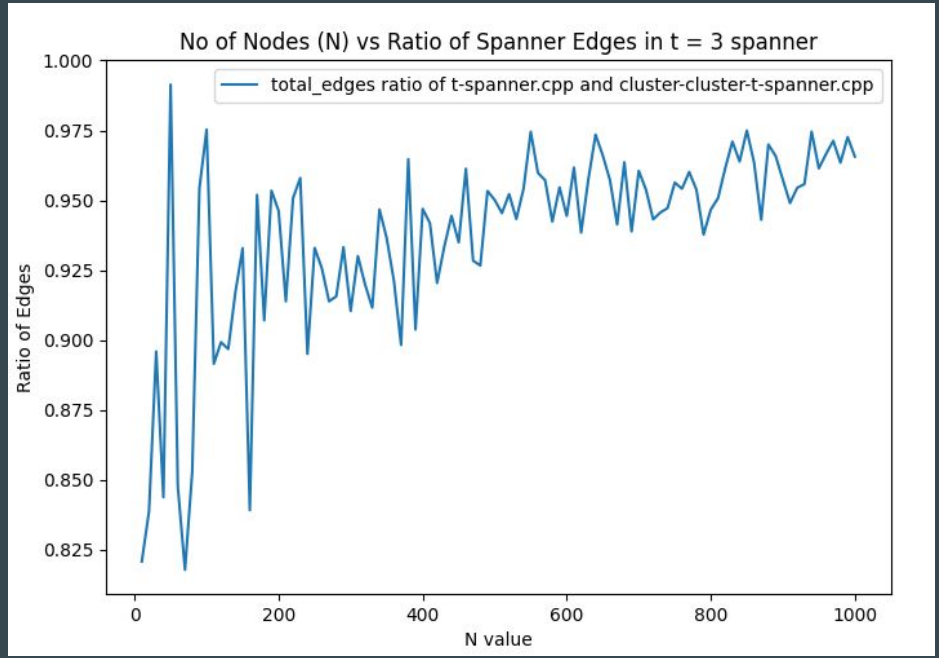
We wanted to investigate where this difference in edges is coming from, so we plotted the number of edges added in Phase-1 and Phase-2 for both the algorithms and found that the difference in edges is not coming from Phase-2 but Phase-1 which makes much more sense as we are running Phase-1 half the times in cluster-cluster.



Test #3: N-value vs Spanner Edges

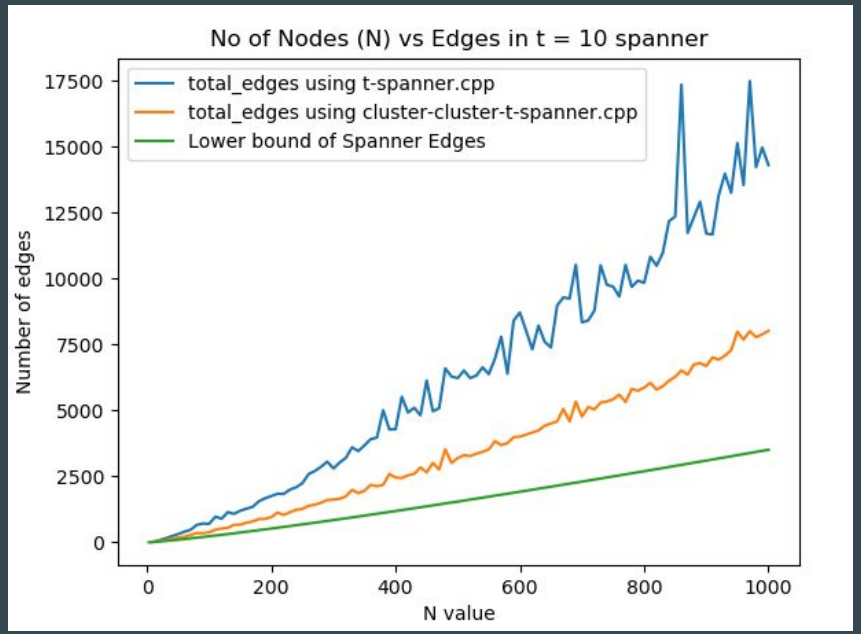
We also wanted to test this decrease in number of edges using cluster-cluster by plotting different number of spanner edges for different n values while keeping t value constant. We started with t = 3 and quickly noticed that there isn't much difference in the number of edges between the two algorithms and the number of edges using t-spanner.cpp is indeed lower than cluster-cluster.

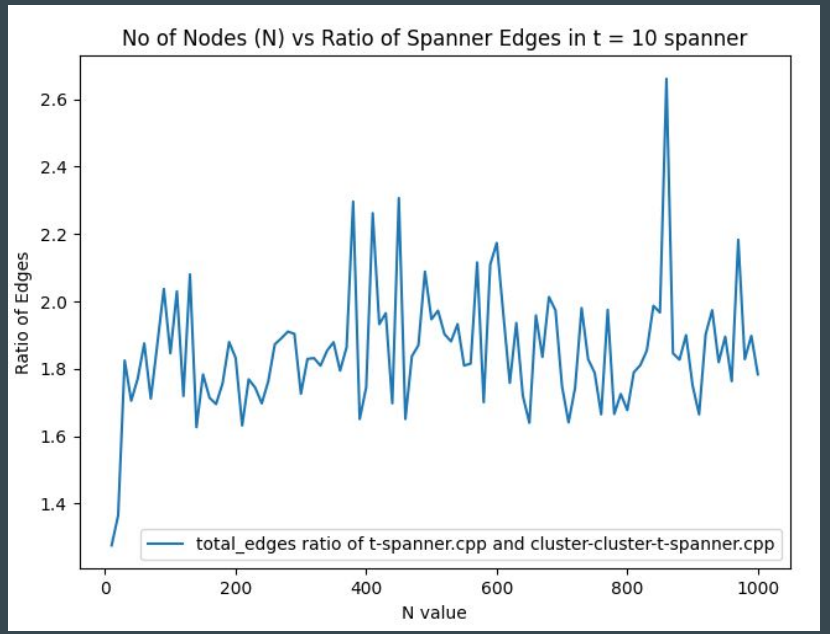




Test #3: N-value vs Spanner Edges - Continued...

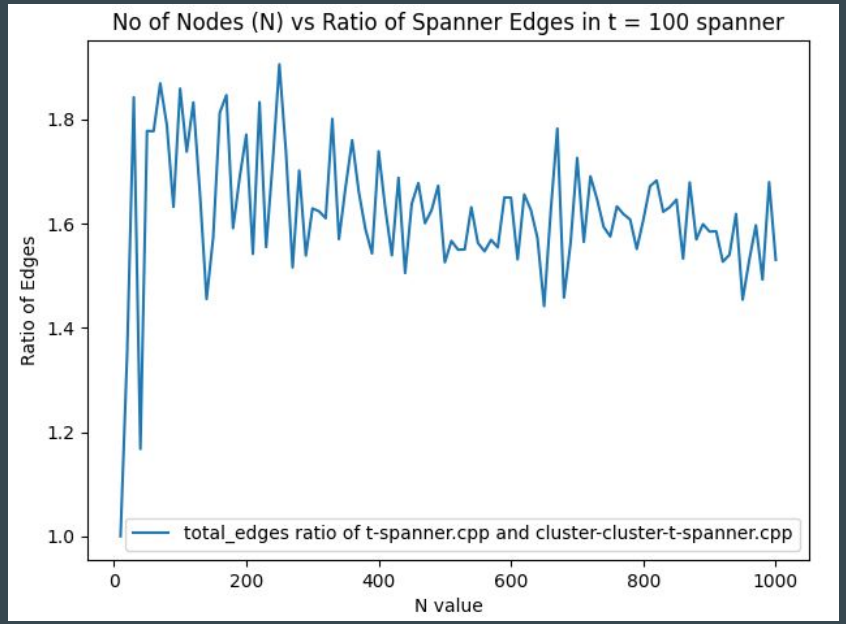
We plotted it for bigger t values. As we go from t = 3 to t = 10, we observe that the number of spanner edges using cluster-cluster is at-least 1.6x better and sometimes even more than 2.0x, reaching a max value of 2.6x at one point.

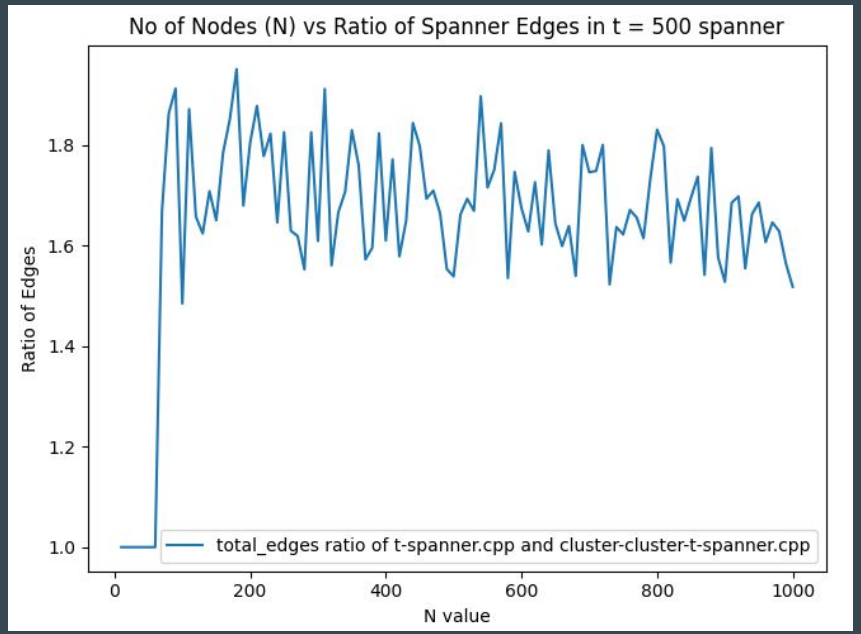




Test #3: N-value vs Spanner Edges - Continued...

As we go from t = 10 to higher values of t like 100 and 500, we notice that the ratio of edges oscillates between 1.5 to 2.





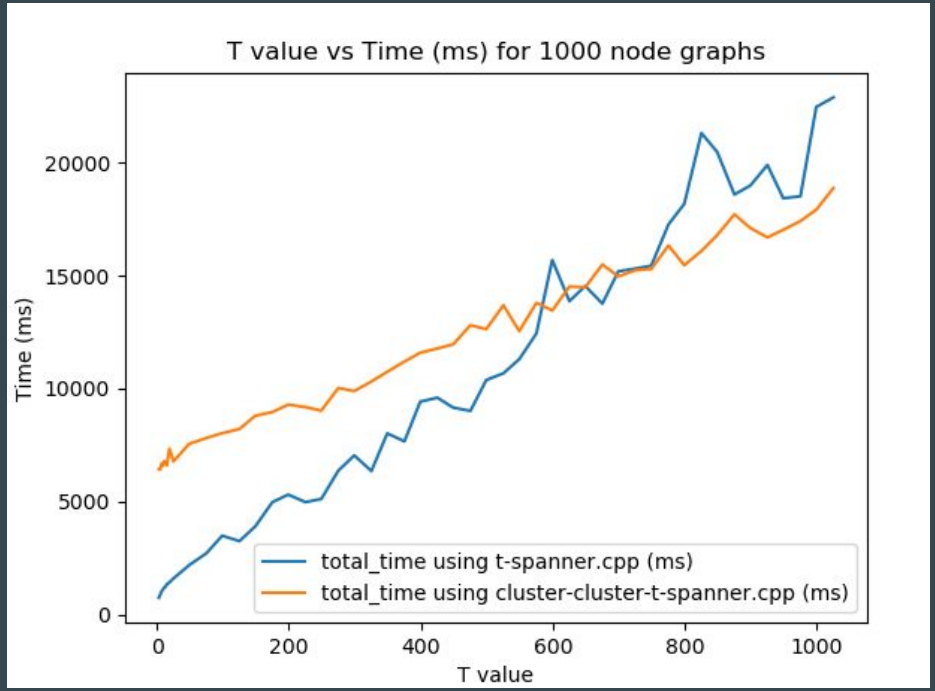
Test #3: N-value vs Spanner Edges - Explanation

- The reason why there wasn't a lot of difference in the number of edges between cluster-cluster and t-spanner.cpp and in-fact cluster-cluster performed worse than t-spanner.cpp for smaller values of t like at t = 3 was because at smaller t values, we are not running Phase-1 of the algorithm enough times.
- For t = 3, both t-spanner.cpp and cluster-cluster will be running Phase-1 exactly once so both the algorithms choose an equal number of edges from Phase-1.

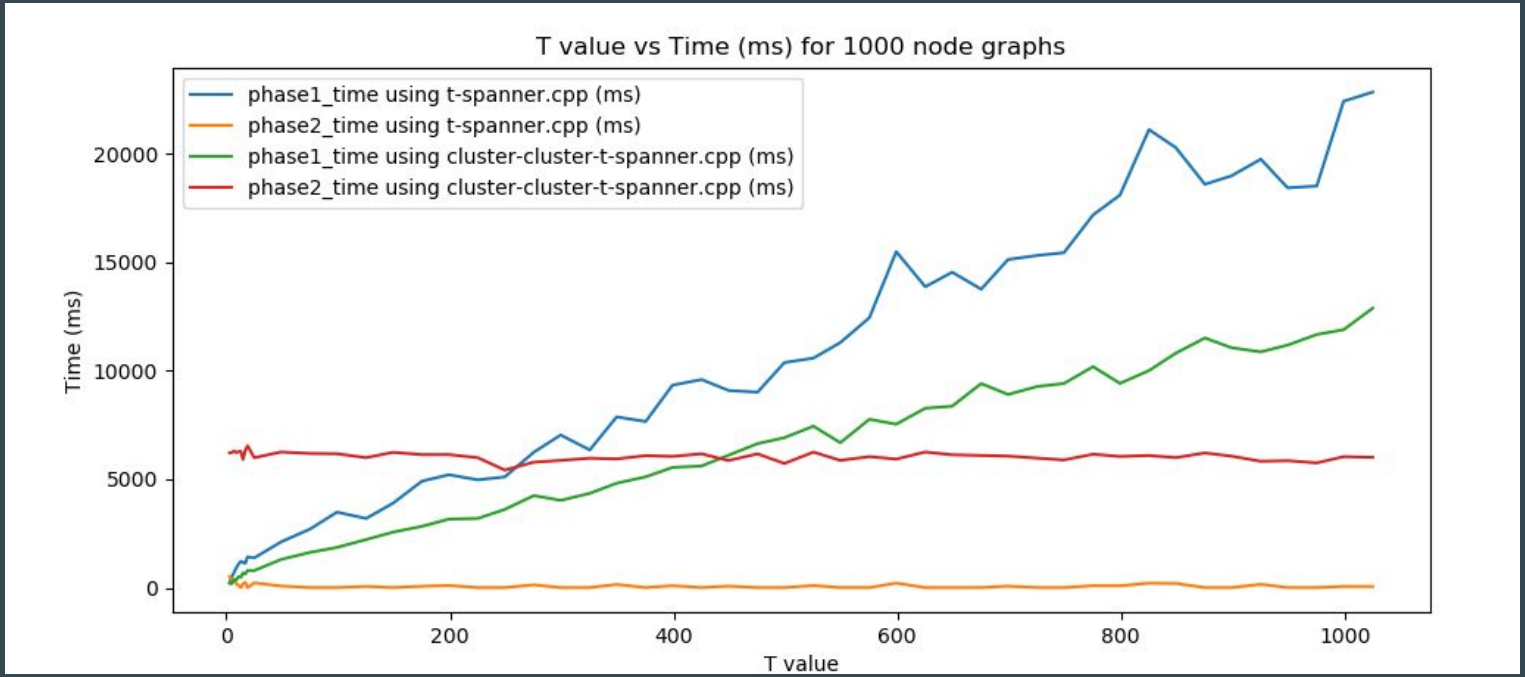
- It is only for bigger values of t that the difference starts because then we are running Phase-1 multiple times and it will be run half the times in cluster-cluster which would also decrease the total edges in cluster-cluster by half.

Test #3: Cluster-Cluster vs T-spanner - Performance

We also wanted to compare the performance of both the algorithms in generating the spanner graph. We started by first plotting the t-value vs time taken to generate the spanner graph for both the algorithms. We quickly noticed that for majority of t-values, t-spanner.cpp took less time than cluster-cluster. To investigate further, we turned to phase wise time taken.



Test #3: Phase-wise Time Taken



Test #3: Phase-wise Time Taken - Explanation

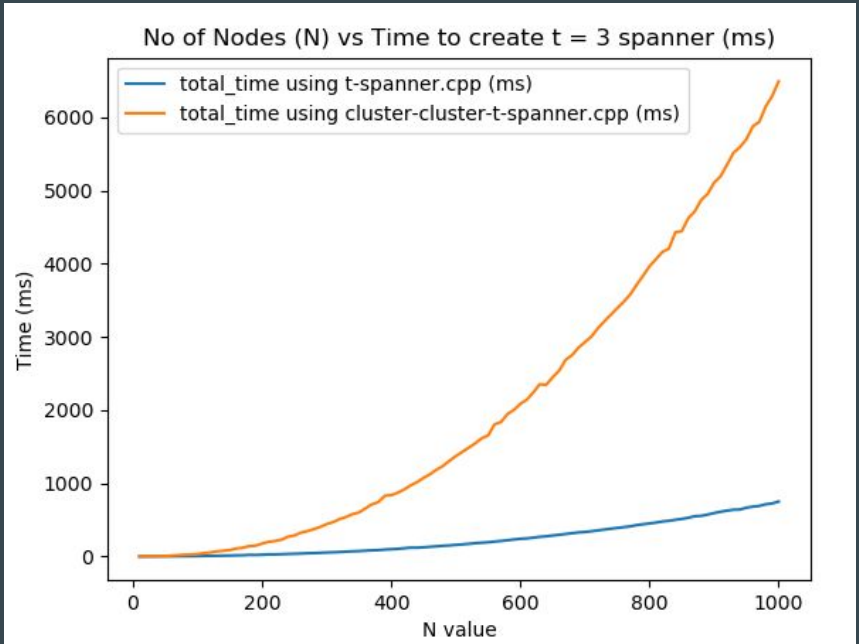
We notice that Phase 2 of "cluster-cluster" takes consistently longer time than t-spanner , however since we run the Phase-1 of "cluster-cluster" half as many times as t-spanner , it spends a lot less time in Phase-1 compared to t-spanner .

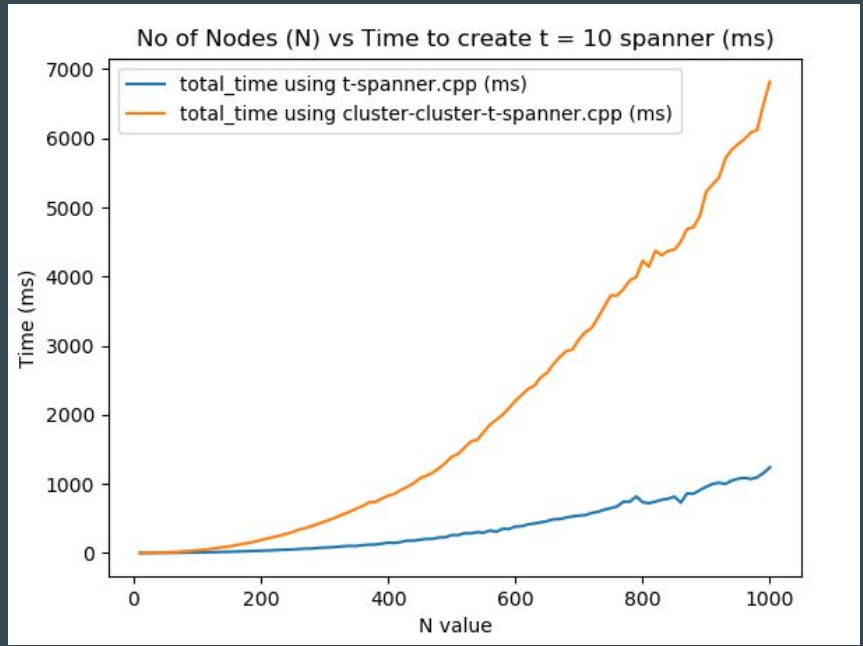
The Phase-1 timings are ideally straight lines and if we fit a straight line to the plots we notice that the slope of Phase-1 of t-spanner is around 2 times the slope of Phase-1 of "cluster-cluster".

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

Test #3: N-value vs Time Taken

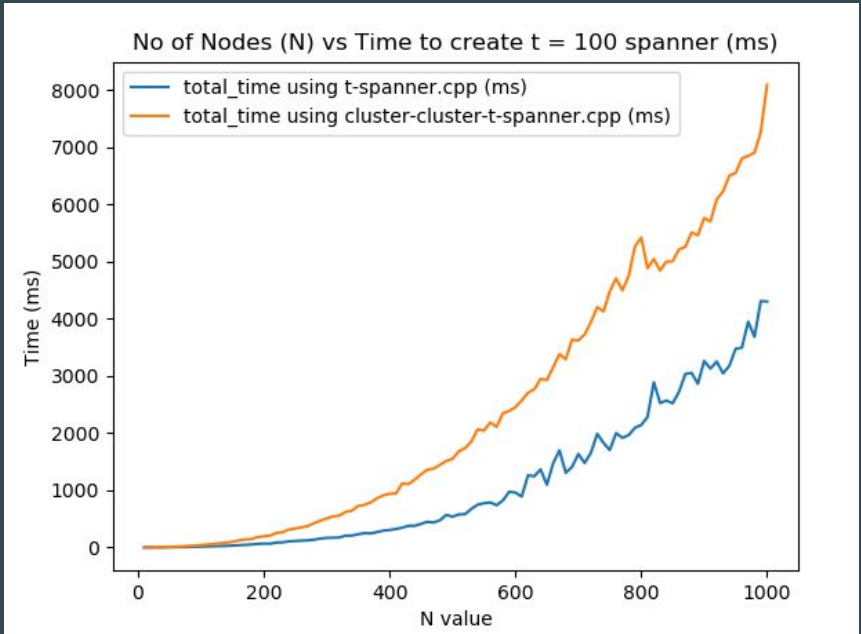
We also wanted to test the performance of both the algorithms for different values of n so we plotted the time taken vs increasing n values. We plotted this for t-values in [3, 10, 100, 500].

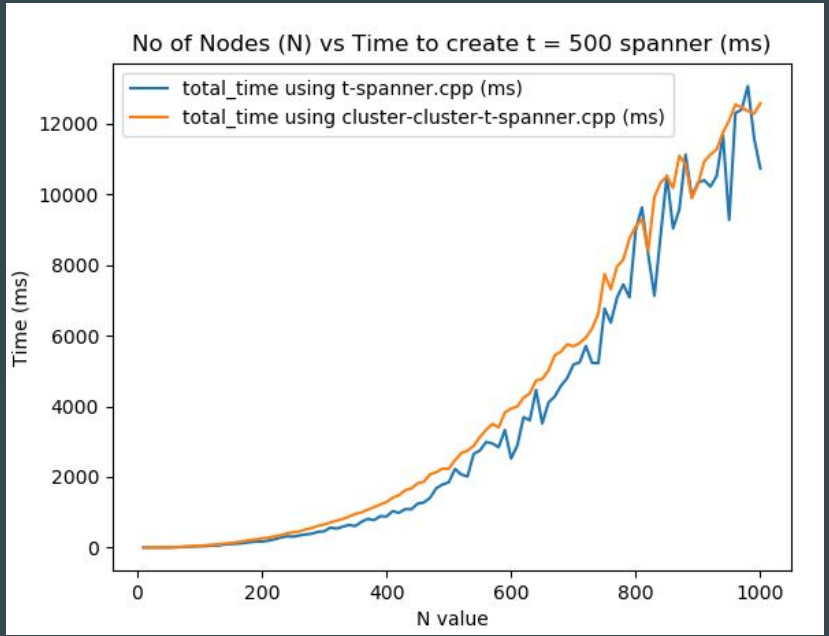




Test #3: N-value vs Time Taken - Continued...

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.





Test #3: Cluster-Cluster vs T-spanner - Conclusion

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, it decreases the edges consistently for t-values larger than or around 10.

Test #4 : Using STL map instead of STL set - Introduction

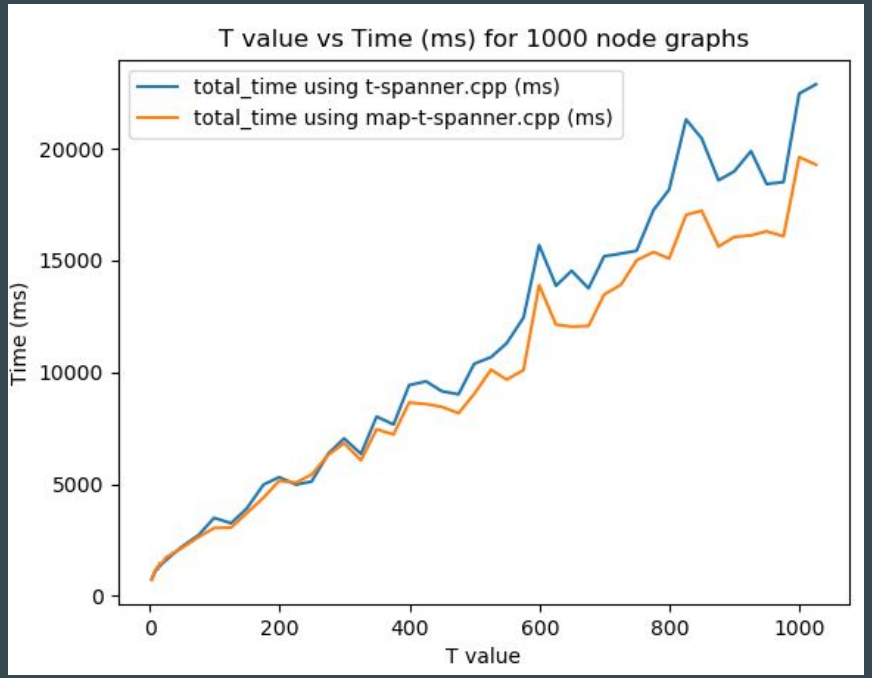
- Sen-Baswana t-spanner algorithm requires a lot of insertion and deletion of edges from the adjacency list.

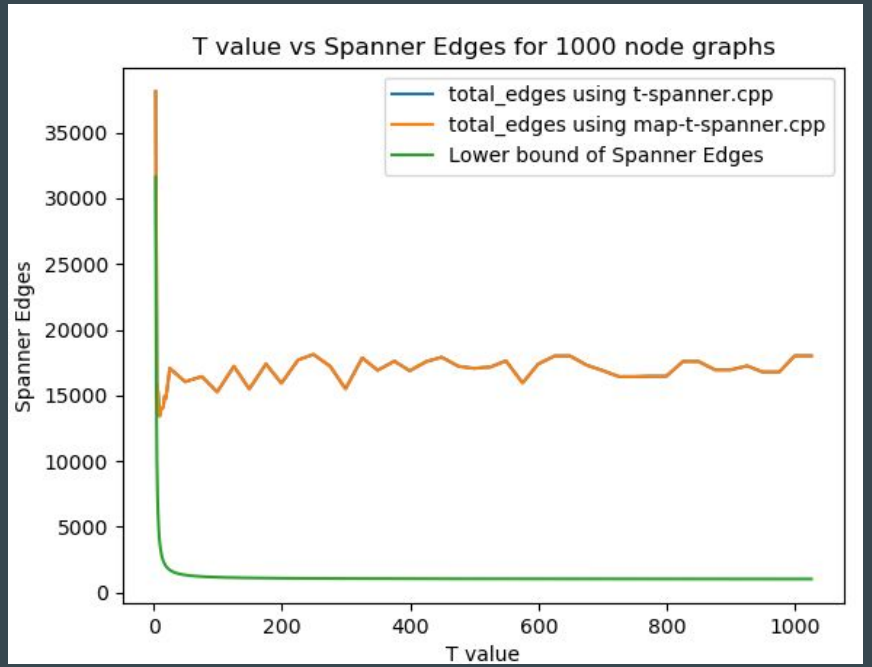
- Vector access time : O(1), insertion : O(1), deletion : O(n)
- Set access time : O(log n), insertion/deletion time : O(log n)
- Map access time : O(1), insertion/deletion time : O(log n) but better constant factor than STL set.

- Typically adjacency lists are implemented in c++ using vectors because we only traverse over the edges in an adjacency list and never delete any edges from it.
- We initially implemented the adjacency list in t-spanner using set. We skipped vector implementation all together because of O(n) deletion time. Next, we wanted to implement the algorithm using map and see if there is any improvement over the time taken

Test #4 : Using STL map instead of STL set

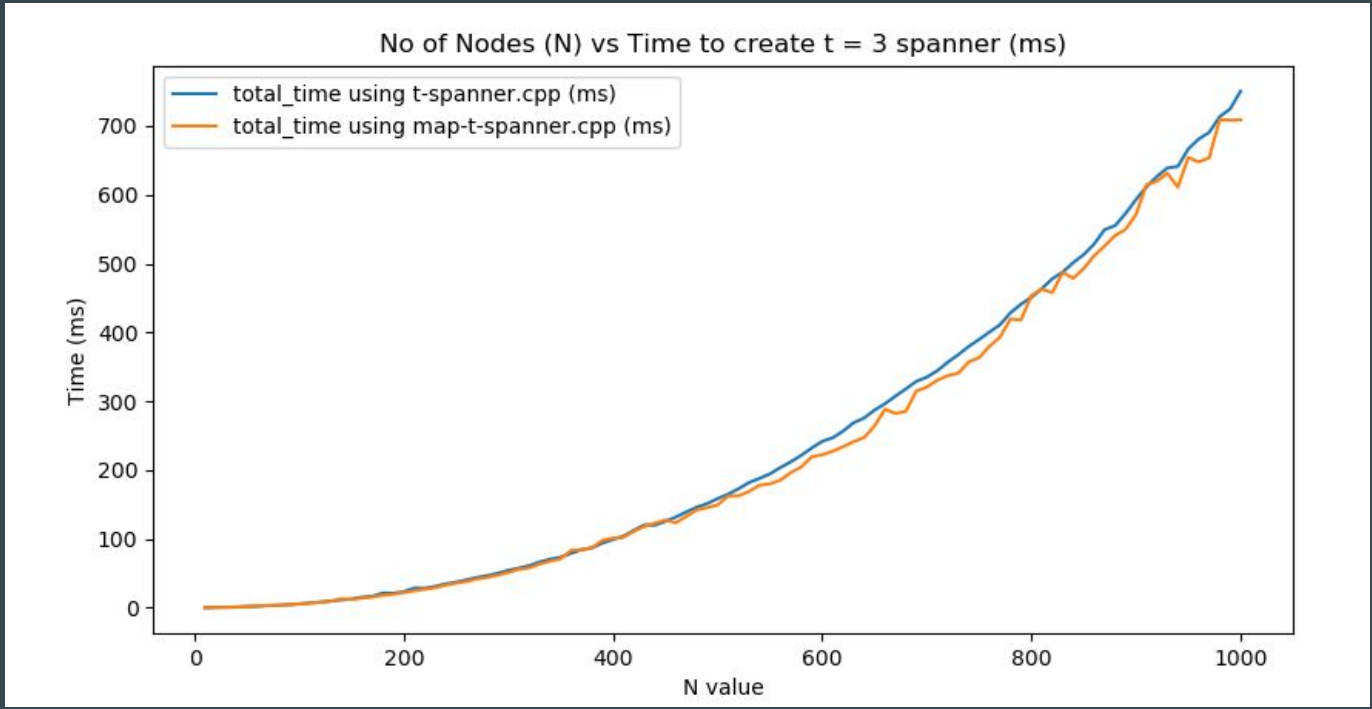
We started by first plotting t-value vs time taken for 1000 node graphs for both the different implementations. We observe that the map implementation is faster for bigger t-values. The number of edges remain the same as expected because we didn't change the algorithm.



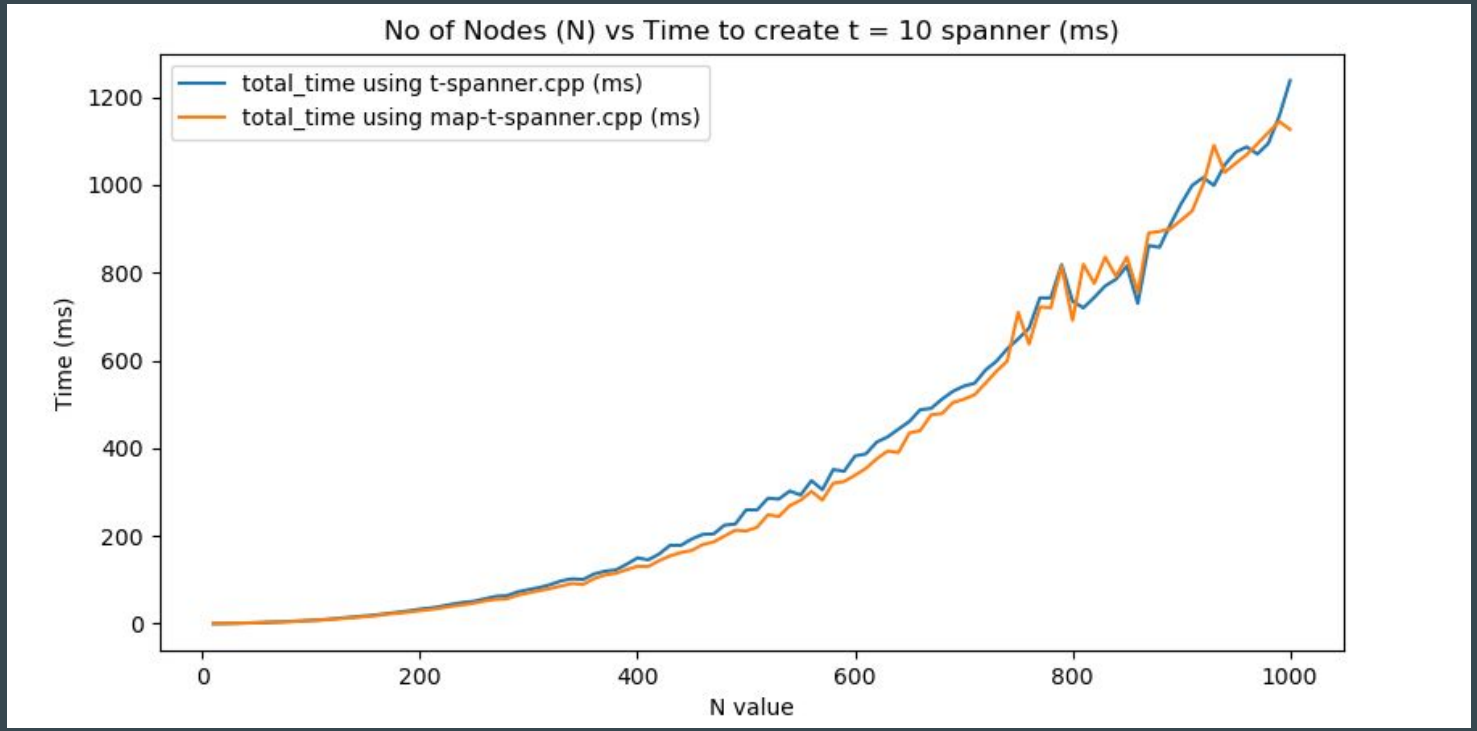


Test #4 : N value vs Time Taken (t = 3)

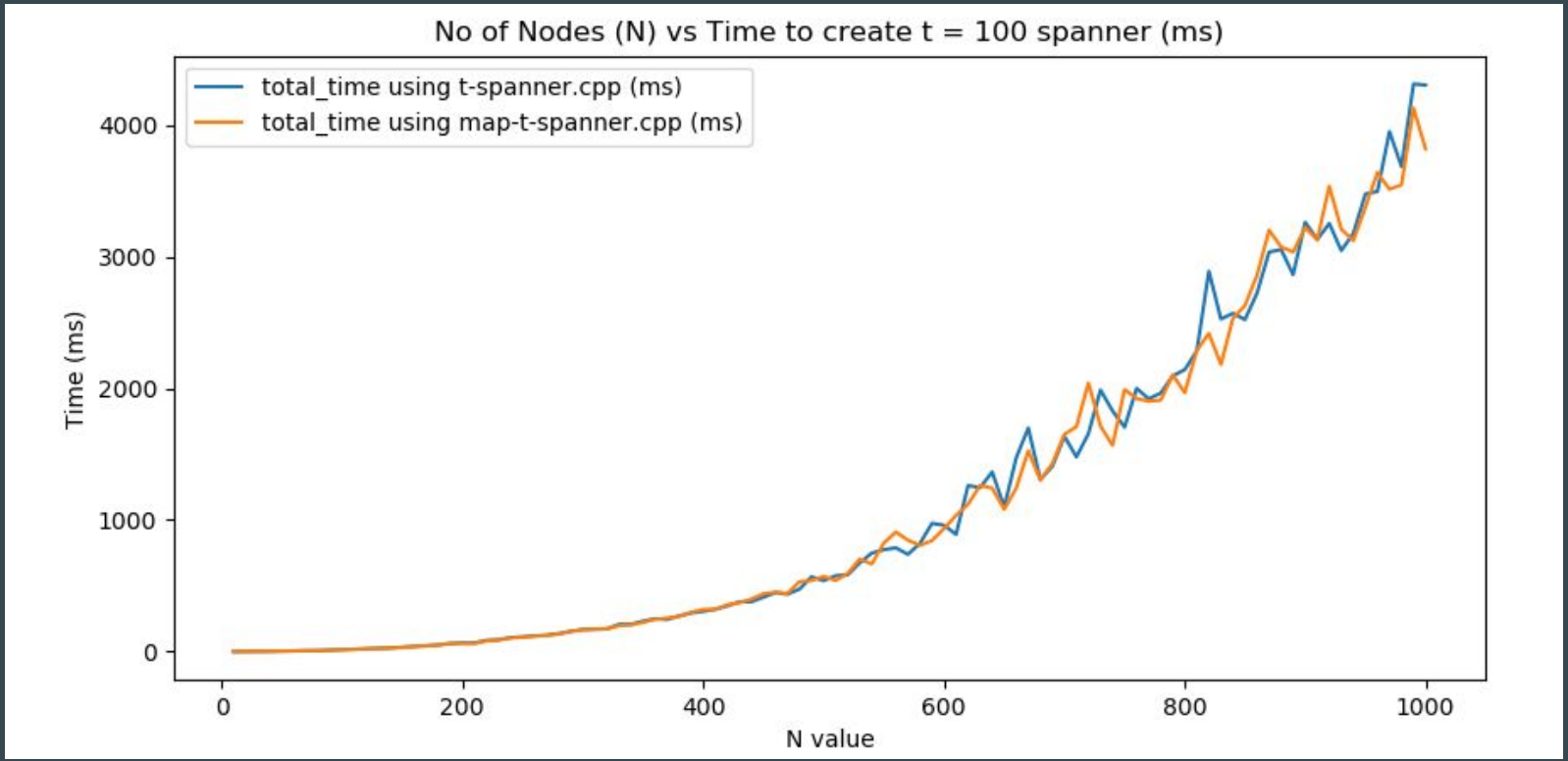
To test further, we plotted the time taken by both the implementation for increasing n-values while keeping the t-value constant. We started with t = 3 and then plotted it for t-values in the range [10, 100, 500]



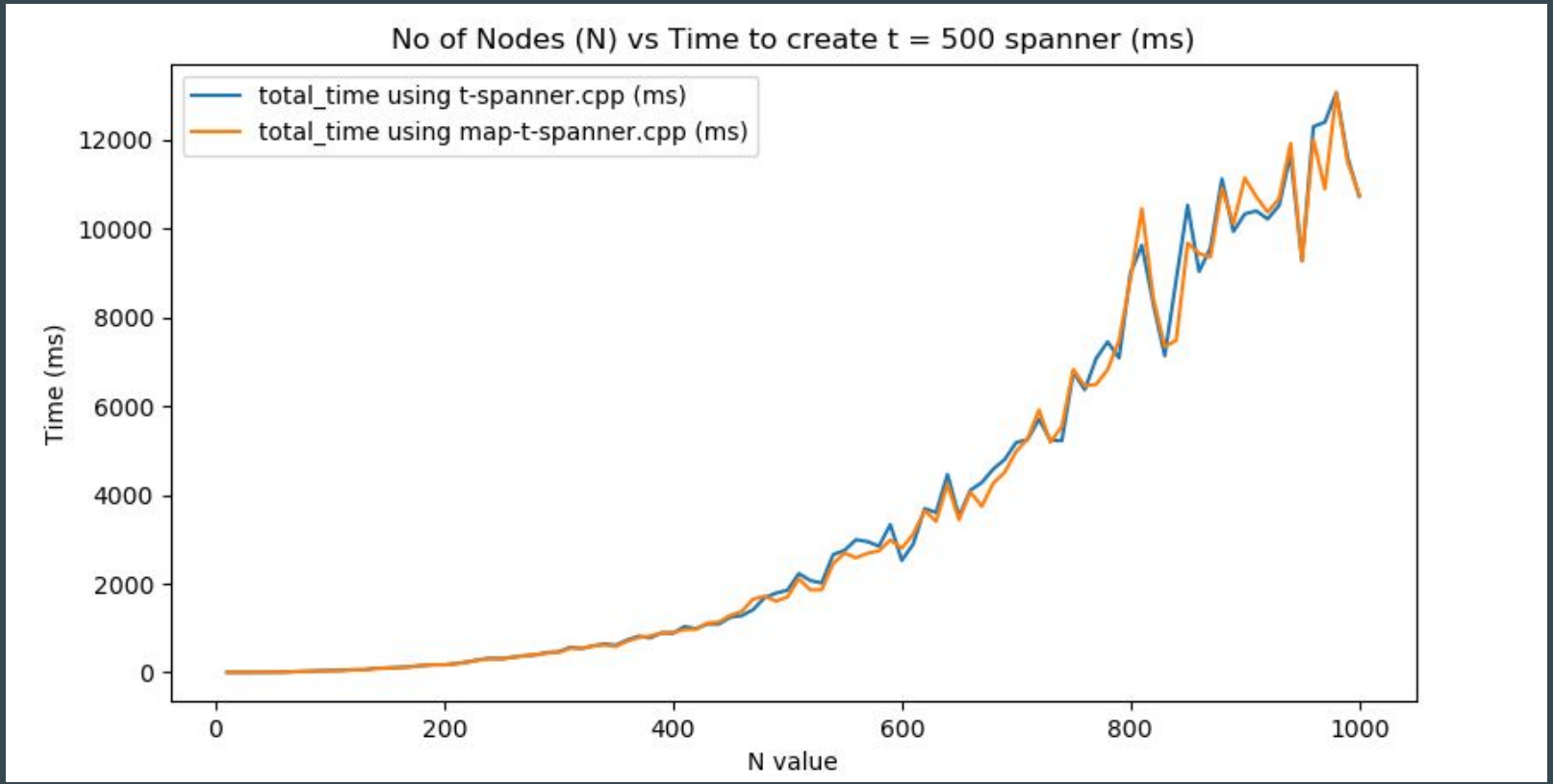
Test #4 : N value vs Time Taken (t = 10)



Test #4 : N value vs Time Taken (t = 100)



Test #4 : N value vs Time Taken (t = 500)

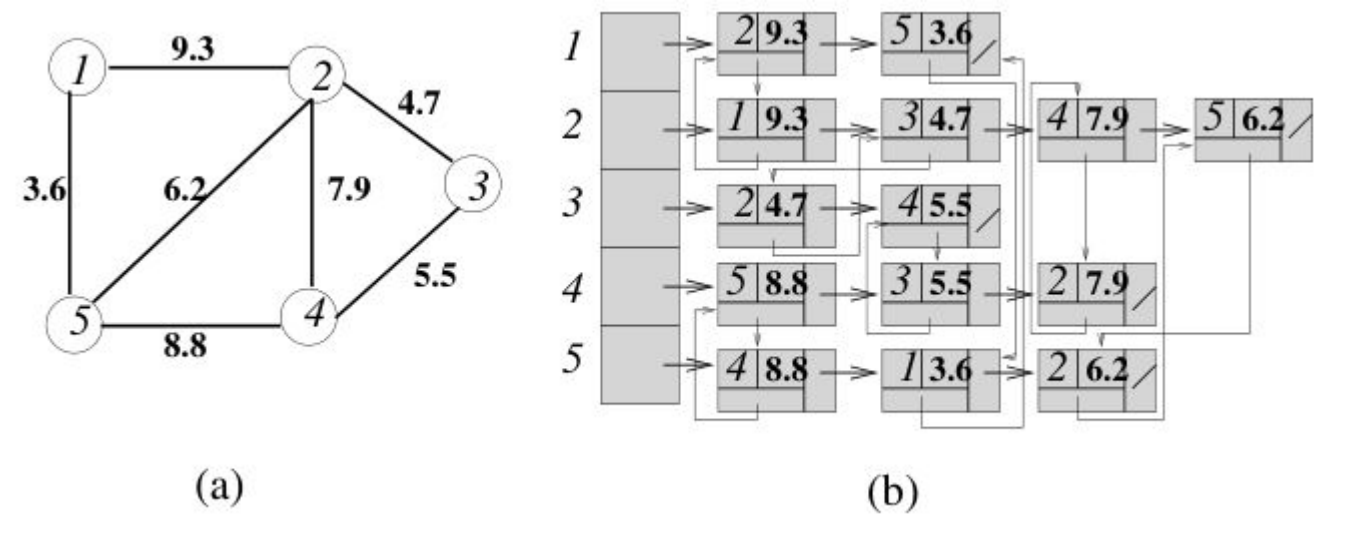


Test #4 : Using STL map instead of STL set - Conclusion

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 which would have O(1) insertion and deletion times as well, 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.

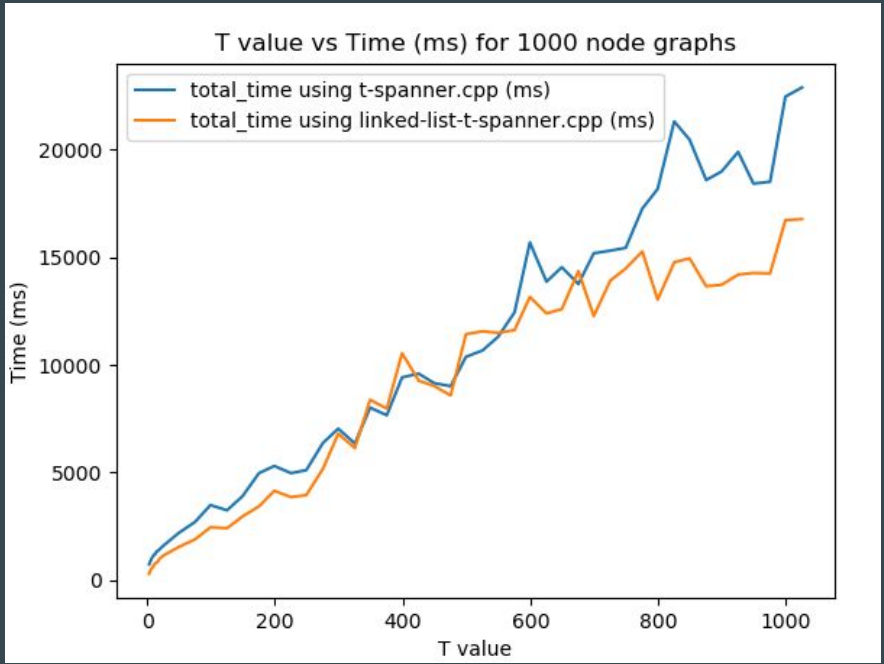
Test #5 : Linked List vs Set/Map implementation - Introduction

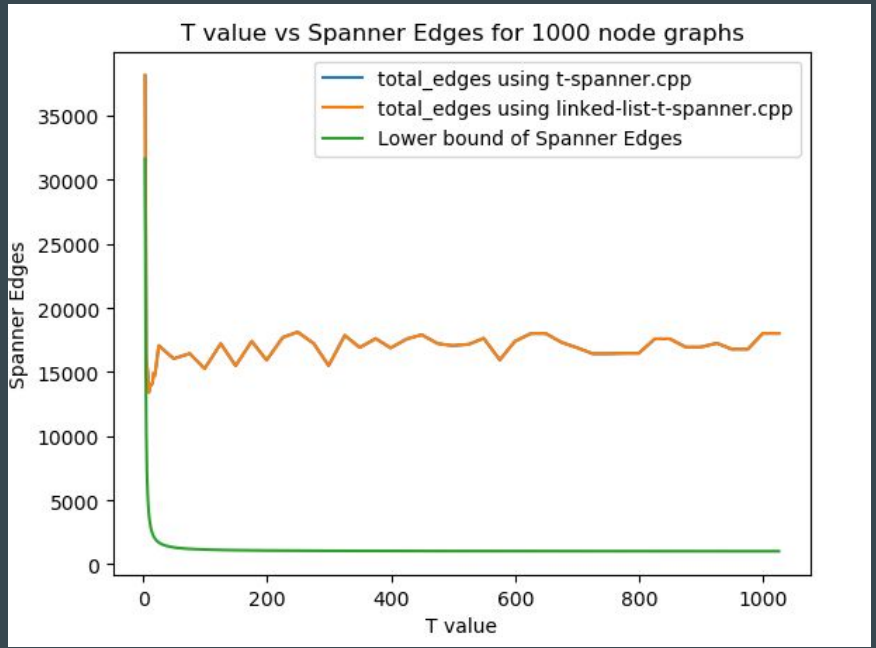
In the paper, the authors suggested another implementation of an adjacency list which they call the augmented adjacency list representation. With this implementation, deletion of an edge from the adjacency list would be O(1). We decided to implement this version of the adjacency list and test the performance against the set and map implementation.



Test #5 : T-value vs Time Taken

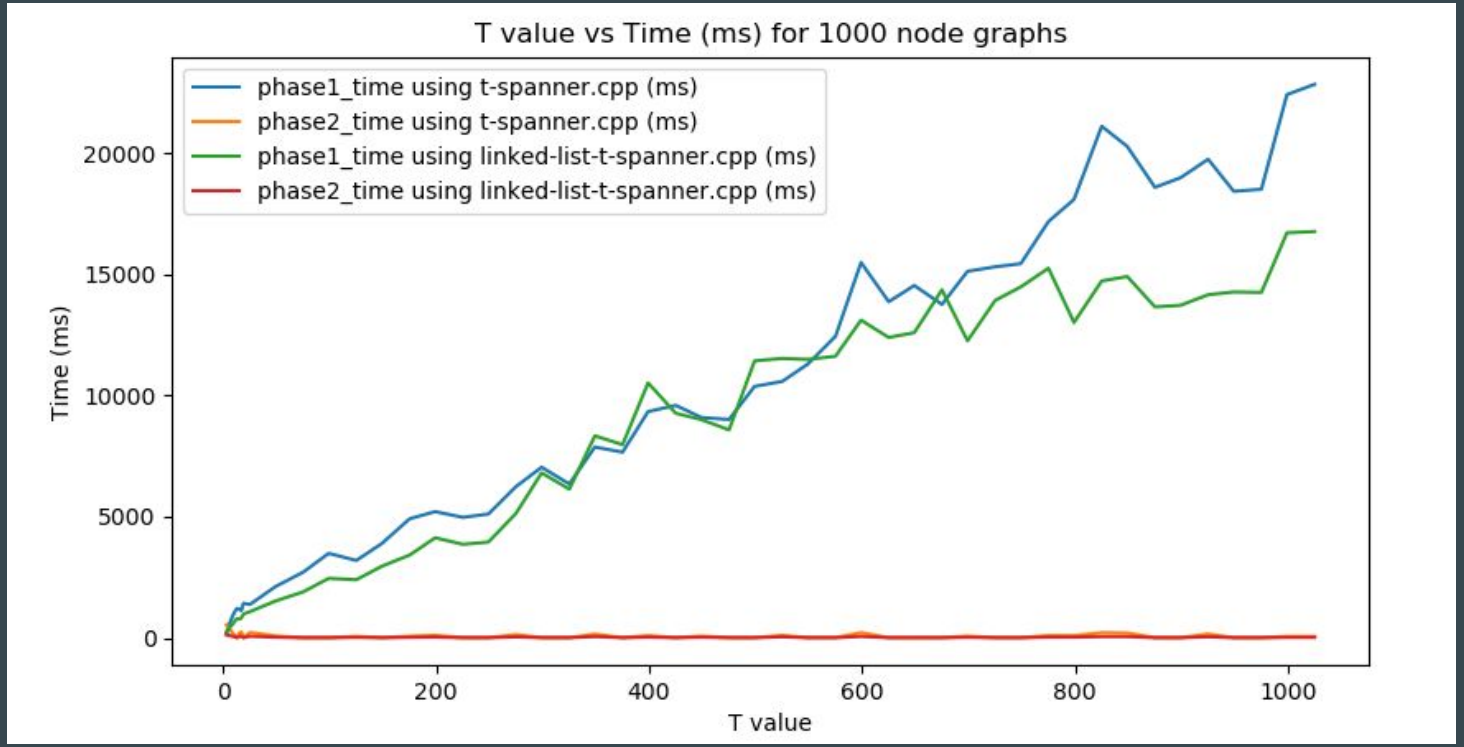
We observe that the number of edges stays the same as expected. However, the time taken is reduced significantly. For large t-values, there is a difference of multiple seconds. Where is the time improvement coming from?





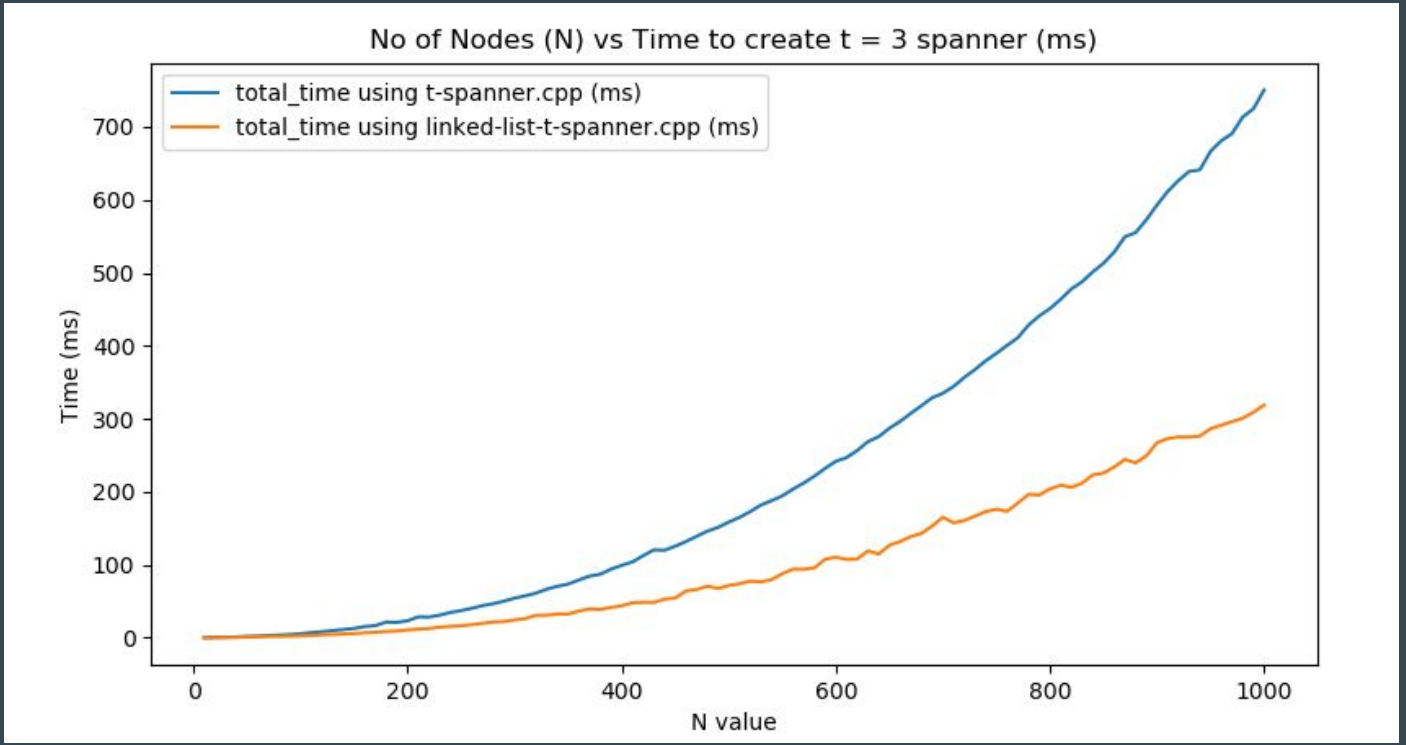
Test #5 : T-value vs Phase-wise time

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.

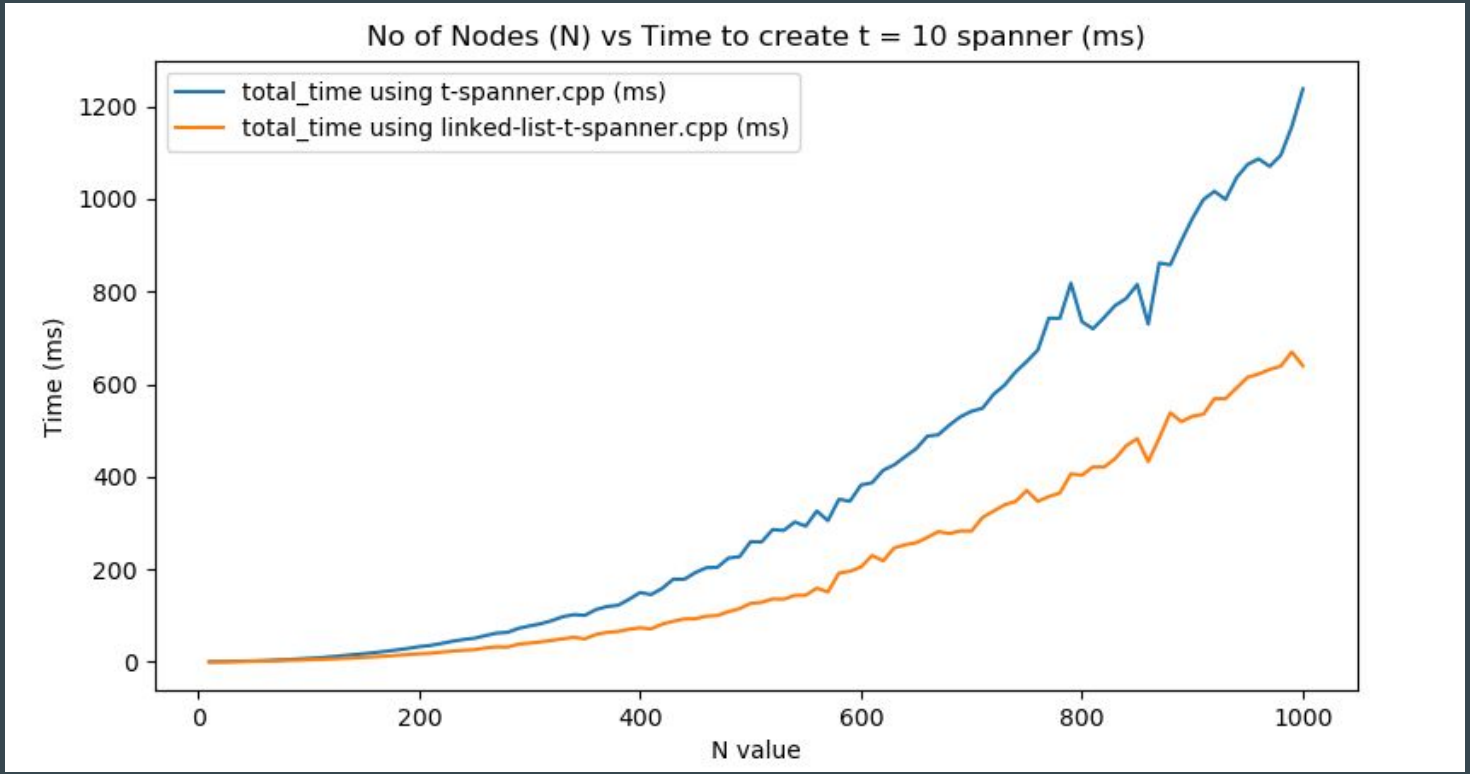


Test #5 : N-value vs Time Taken (t = 3)

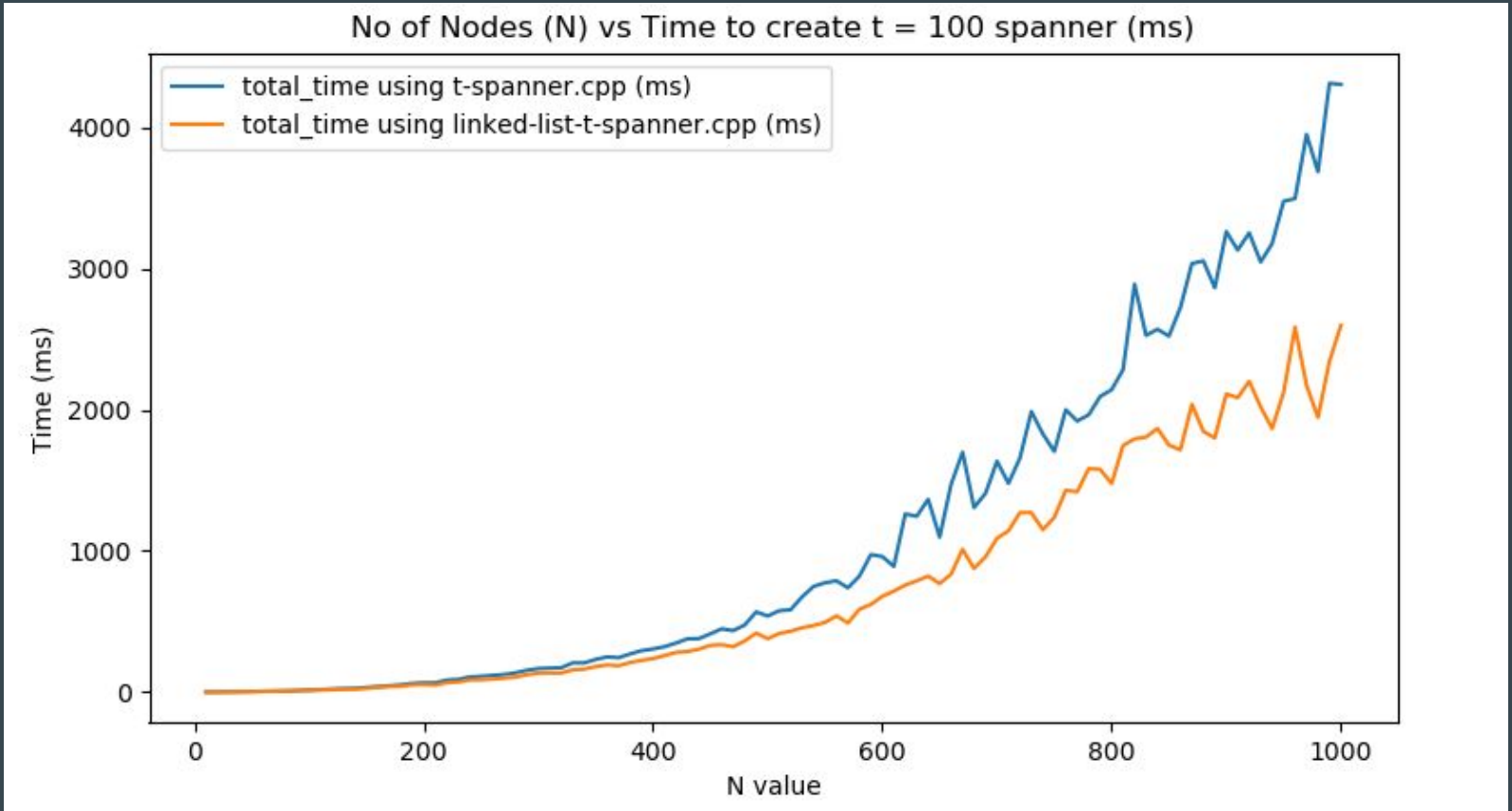
To test further, we started plotting the time taken by different implementations for increasing values of n while keeping the t-value constant. For t = 3, we observe that there isn't much of a difference for smaller values of n, but it becomes quite noticeable for bigger n values.



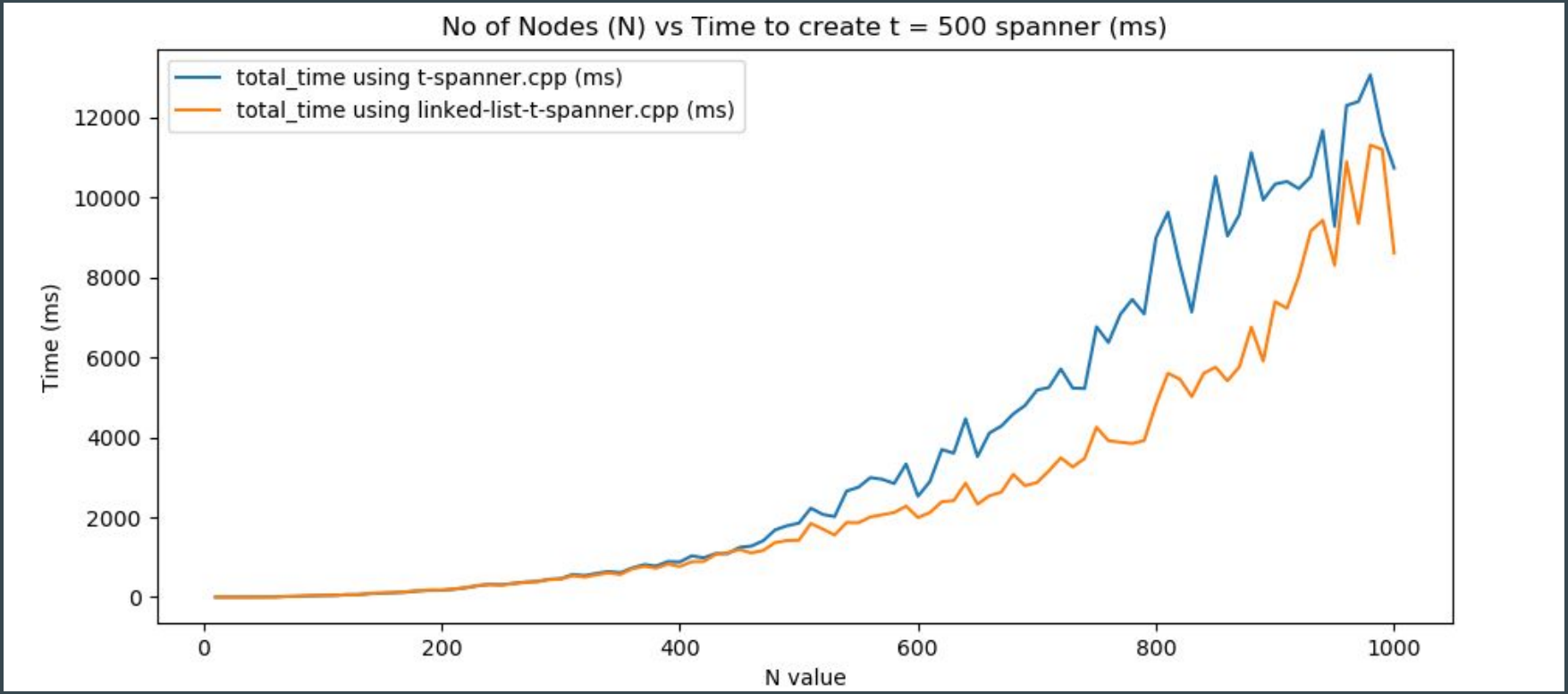
Test #5 : N-value vs Time Taken (t = 10)



Test #5 : N-value vs Time Taken (t = 100)



Test #5 : N-value vs Time Taken (t = 500)



Test #5 : Linked List vs Set/Map implementation - Conclusion

From all the plots, we can see that the linked list implementation performs significantly better than all the other implementation (set and map). The difference isn't much for smaller values for n and smaller values of t but it becomes significant and quite noticeable for bigger values of n and t.

- For smaller values of n, the O(log n) deletion factor of set and map implementation is not that significant because of which there is not much of a time difference between the two. As we increase n values, this O(log n) factor starts becoming significant and slows down the set and map implementation leading to a time difference between them and the linked list implementation
- For smaller values of t, we are not running Phase-1 many times so even if n-value is big, i.e. O(log n) value is big, there is not much of a time difference (around 300-400 ms at t = 3). As we increase t values, we are running Phase-1 more and so many more edges are getting deleted from the original graph and as each deletion is O(log n) in other implementations, that slows them down and we see a significant time difference (around 2-4 s at t = 100, 500).

Test #6 : Non-Random implementation - Introduction

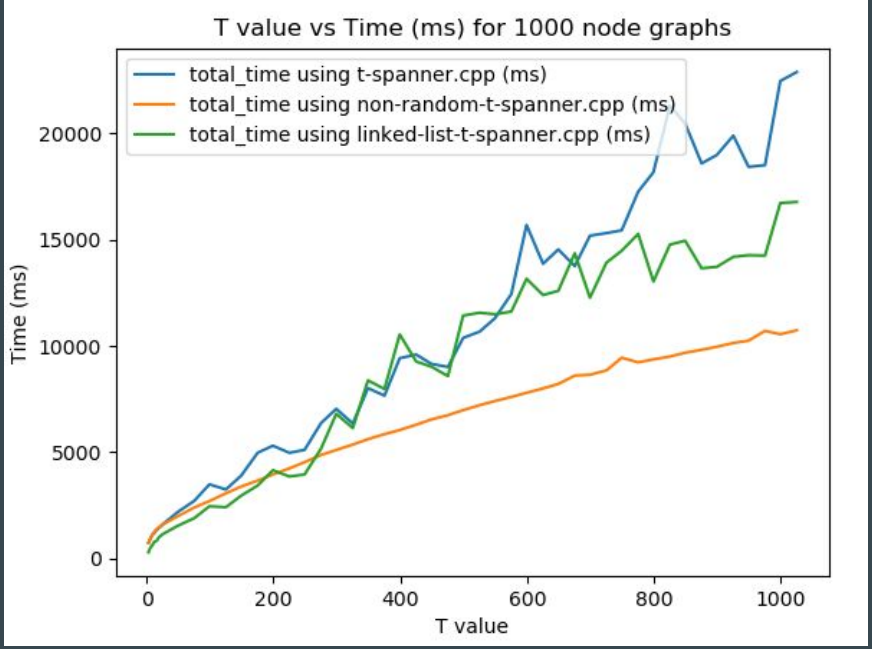
We notice that we generate random number for every cluster in each iteration of t-spanner. These random numbers are used in sampling the cluster in each iteration. We wanted to try picking the first “S” clusters deterministically in each iteration, here “S” denotes the expected number of clusters to be selected in each iteration.

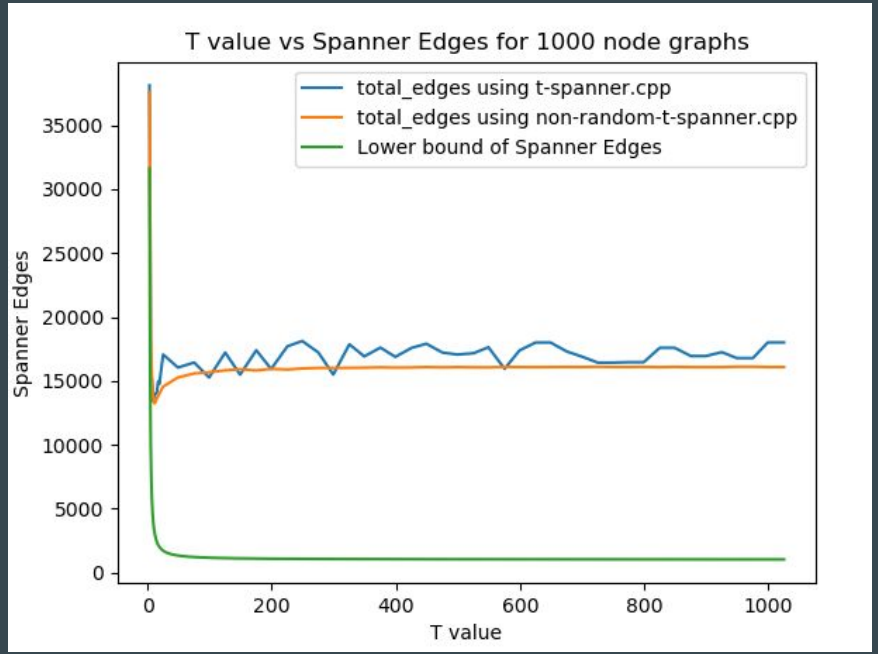
- The probability of selecting a cluster is n^(-1/k)
- If there are C clusters in (i - 1)th iteration, then there will C * n^(-1/k) expected clusters in the ith iteration
- As there are n clusters initially (i = 0), so there will be n^(1 - i/k) number of expected clusters in the ith iteration.

As we are not generating random numbers in each iteration anymore and we know that generating random numbers is quite computationally expensive, so we hypothesized that the non-random implementation would take much less time than the original implementation.

Test #6 : T-value vs Time Taken

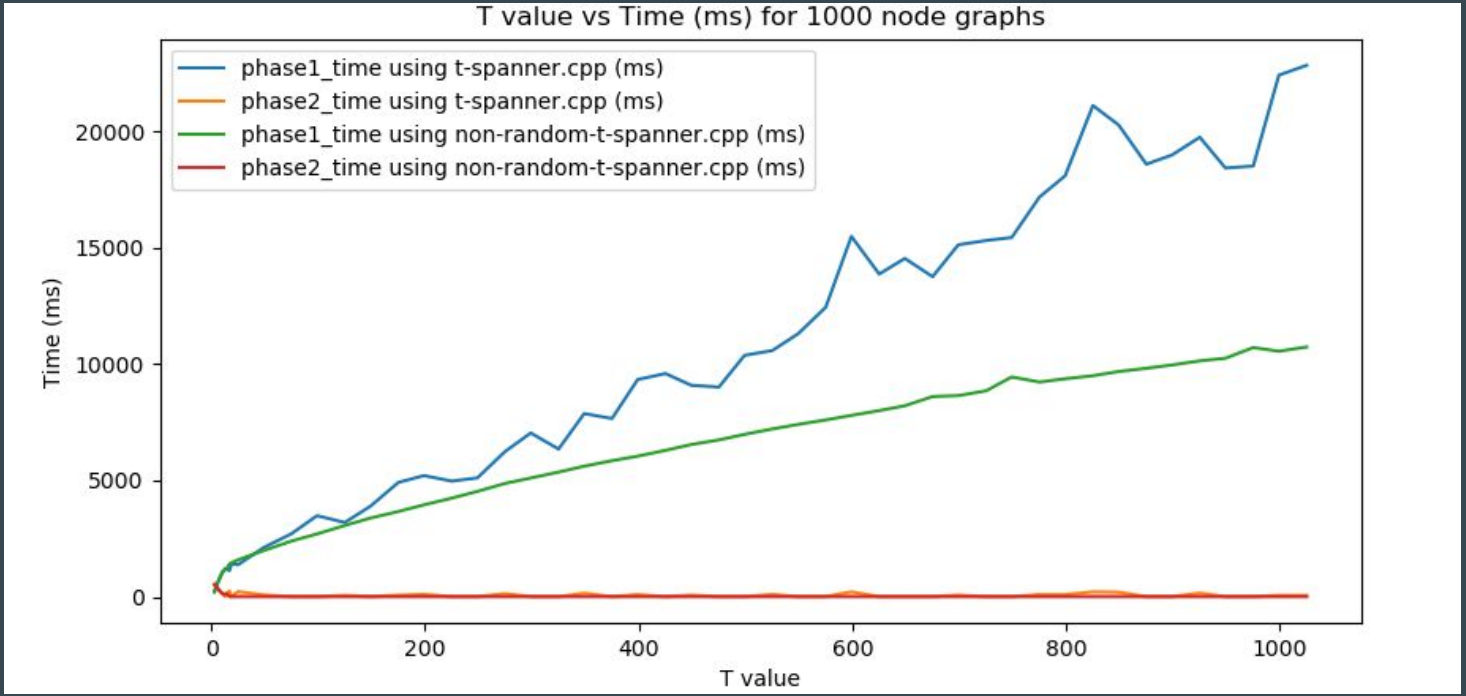
We observe that for 1000 node graphs and increasing t-values, non-random performs significantly better than other implementations and there is a significant time difference for large t-values. We were also surprised to find that linked-list performs worse than non-random which uses the set implementation as before and also non-random decreases the total edges as well.





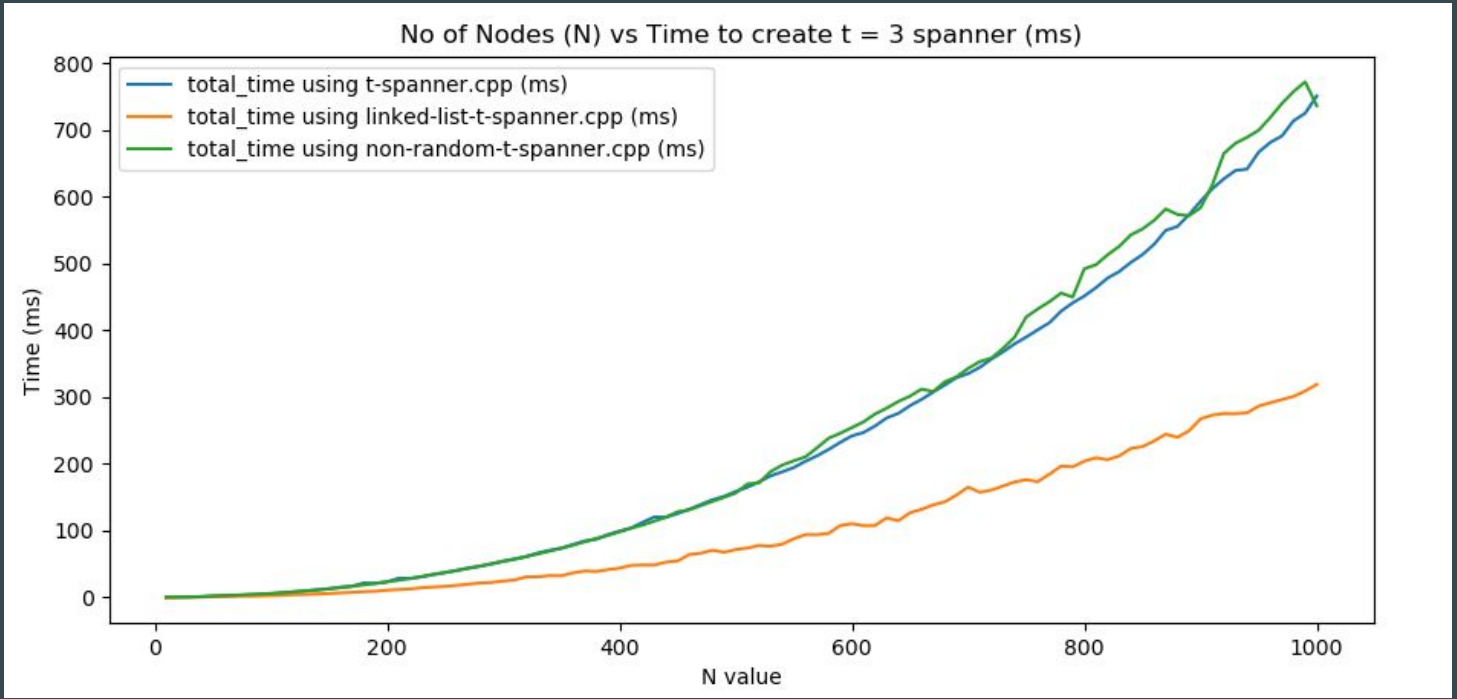
Test #6 : T-value vs Phase-wise time

As we can see, Phase-2 for both the implementations takes the same amount of time, but there is a significant improvement for time taken by Phase-1. This shows that our hypothesis that non-random would be better is correct but we still don't know whether this difference is coming because of not generating random numbers or some knock-off effect due to always sampling deterministically.

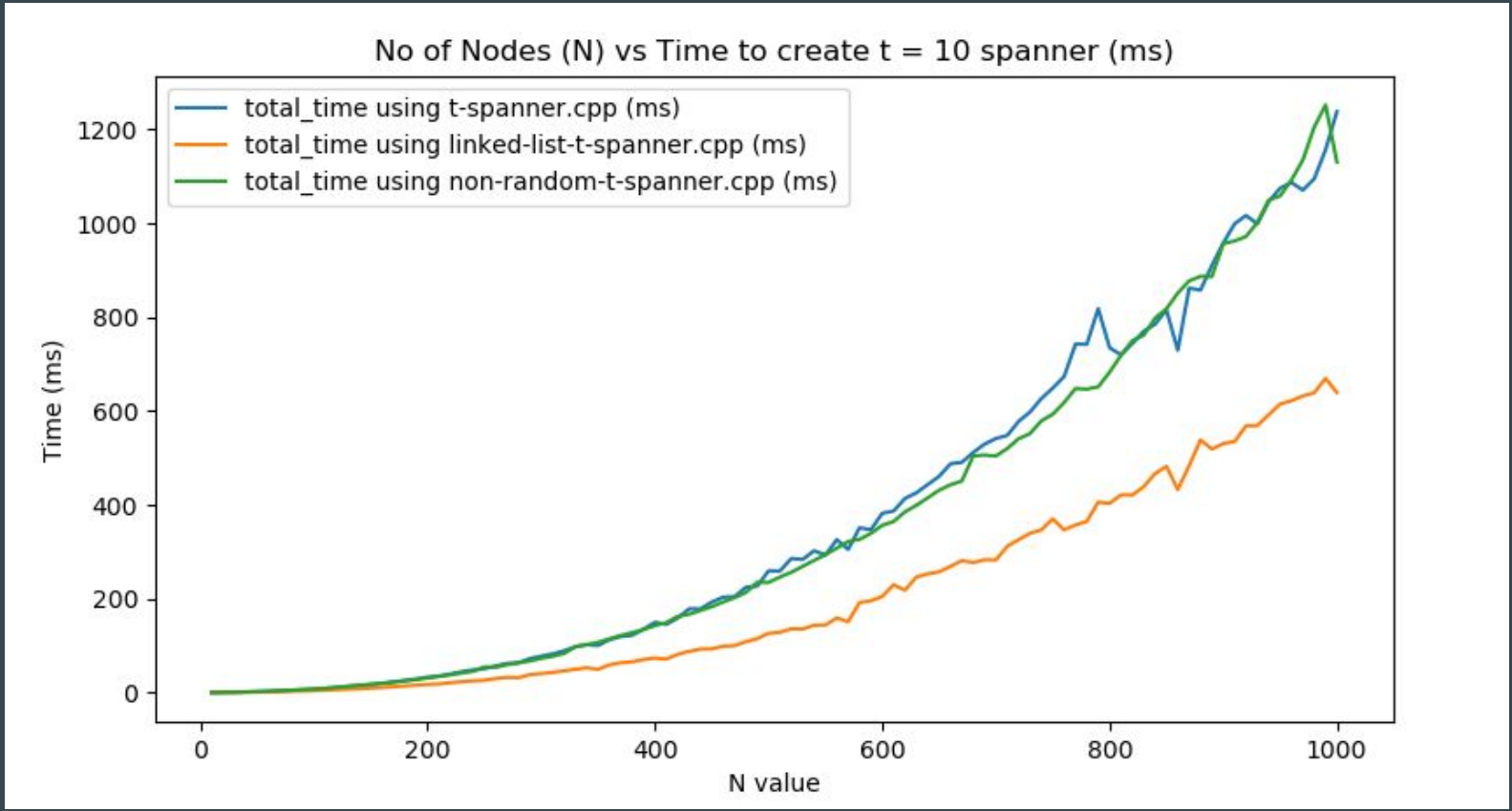


Test #6 : N-value vs Time Taken (t = 3)

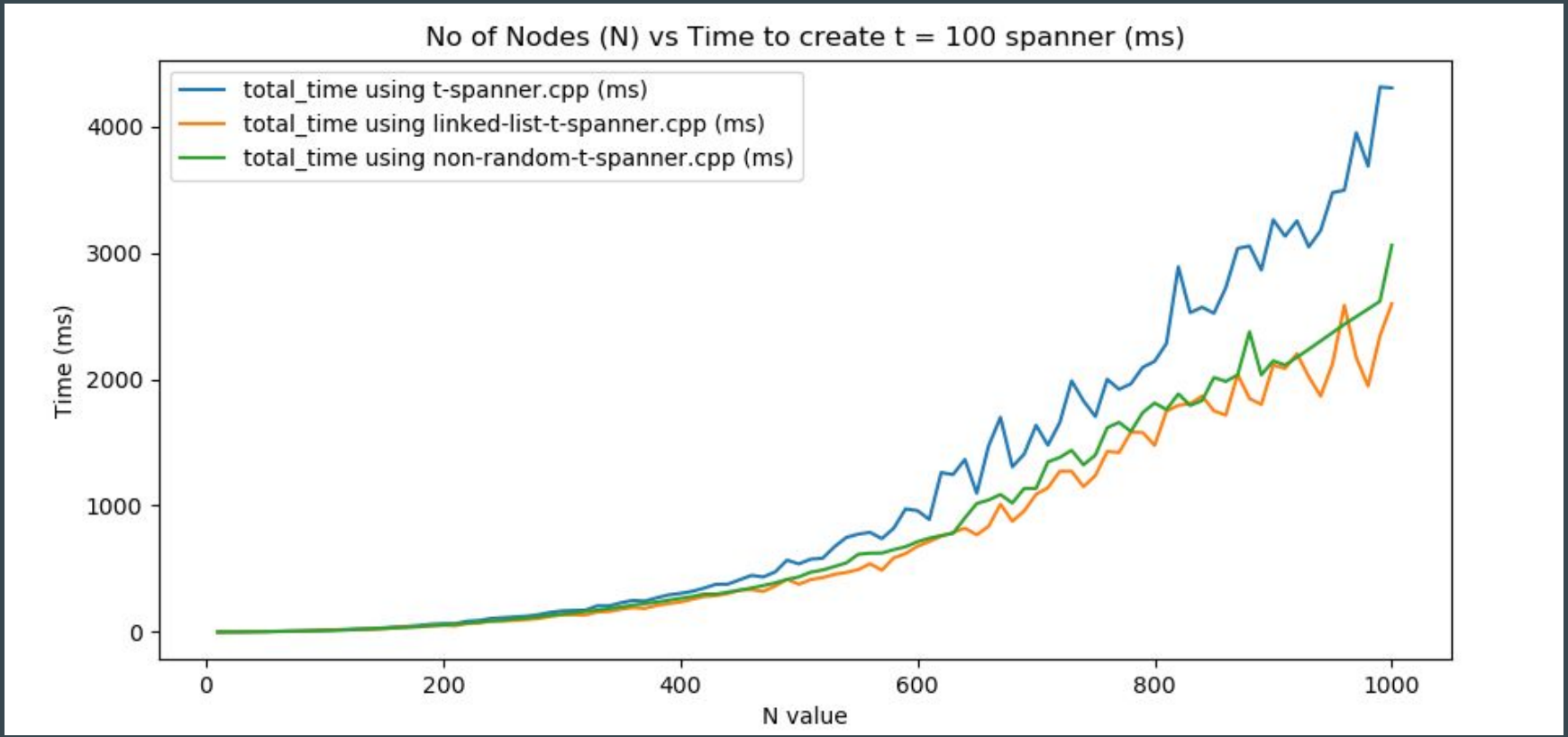
To compare further, we started plotting the time taken by non-random and our fastest implementation (linked list) for increasing values of n while keeping the t-value constant. The results were quite interesting...



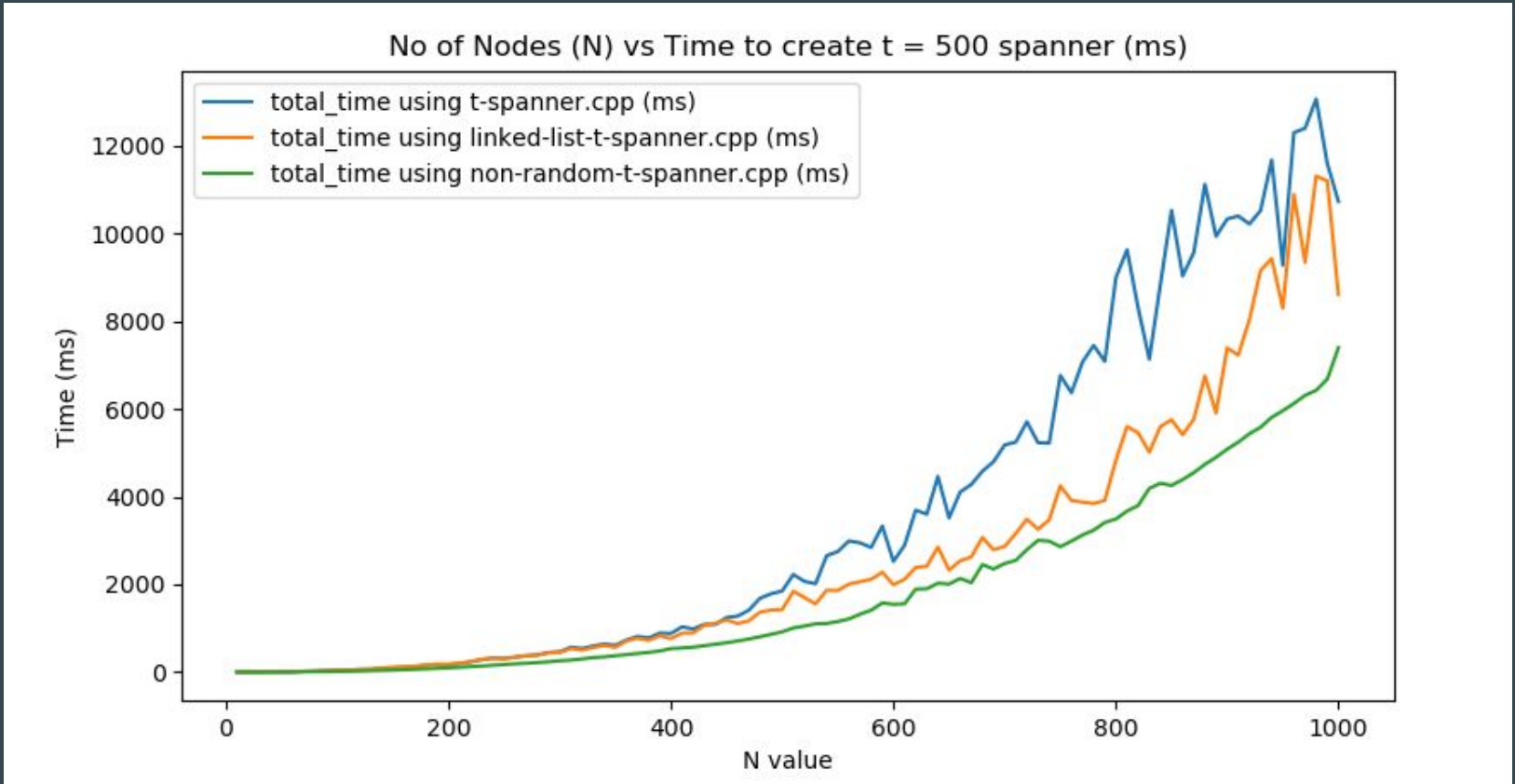
Test #6 : N-value vs Time Taken (t = 10)



Test #6 : N-value vs Time Taken (t = 100)



Test #6 : N-value vs Time Taken (t = 500)



Test #6 : Non-Random implementation - Conclusion

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.

Note: 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.

T-spanner Construction : Conclusion

- For smaller values of t (like t = 3), the base t-spanner algorithm with linked list implementation is the fastest and produces the least 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.

- 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.

T-Spanner Construction : Future work

There were a lot of other things that we wanted to try out and but didn't get the time to test out so we are make a list here for what other things can be done :

- Find the exact reason why non-random implementation is performing so much better in terms of time
- Implement non-random linked-list cluster cluster version of the algorithm and test it against all the other implementations
- Tweak the graph generator, trying out different distributions for selecting the edge weights.
- Testing on much bigger graphs (n > 10000).
- Score the graphs made using different generators and find which distribution is best to select edge weights and why.
- Implement the parallel and distributed versions of the algorithms and test them against the current implementations.
- Find algorithms to support updates to the spanner graph whenever the original graph is updated, i.e. - online updates to the spanner graph.

Graph with all the implementations