Heng Zhou
Hongyang Zheng

Programming Assignment 2

ANLY550
April 13, 2019

First part

Introduction

The code is shown in *randmst.py*.

The code with time calculation is shown in $randmst - time.py$.

In this project, we implemented Kruskal's algorithm to generate a minimum spanning tree (MST). We run the algorithm in Python with different dimensions including 0,2,3,4 and different number of nodes n including 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384. Since it needs to wait for a long time for large number of nodes > 8192: for example, when n=16384, it costs us about 20 minutes for each dimension 2, 3, 4. We only use number of nodes no larger than 16384 and the minimum number of trails 5 in our algorithm. Then we plot the average weight of MST respect to n and how the weight can grow as a function of n in RStudio.

Results under each Dimension

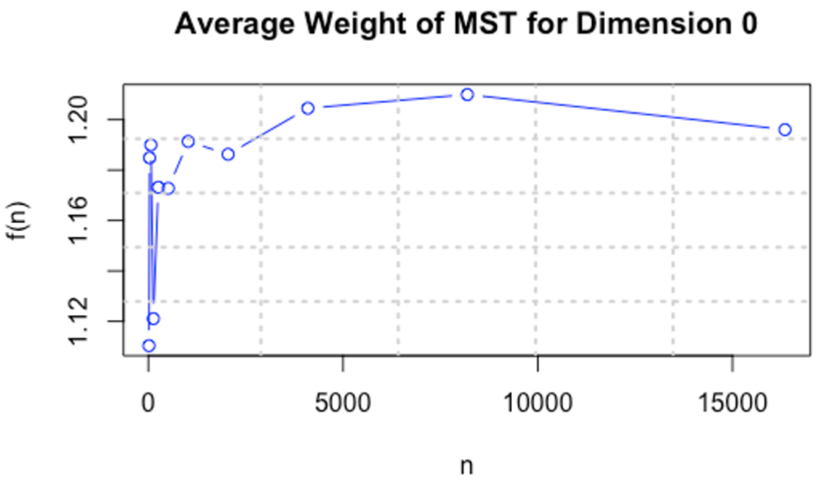

Figure 1: Average Weight of MST

The graphs here describe how average weight of MST change respect to n and how it grows as a function of n. First we have a graph shows average weight of MST in dimension=0. When n is small, the average weight is unstable, it fluctuates dramatically. When n increases, it stabilizes to around 1.2, so the function here is $f(n) \approx 1.2$.

When dimension = 2, we have two graphs, Figure 2 describes the average weight of MST respects to change in n. After plotting the graph average weight of MST w respect to n, we saw a function looks like very similar to $f(x) = x^c$, where $0 < c < 1$, so we first converted n into $z = n^{0.5}$ and then built a linear model between w and z to see whether our assumption is correct. The summary of the model shows that the R^2 for the model is 1, which indicates that this model is good for estimating the relationship between n and w. We got coefficients from the model summary, which gives the function $f(n) = 0.05877 + 0.56916n^{0.5}$. Figure 3 shows the function $f(n)$.

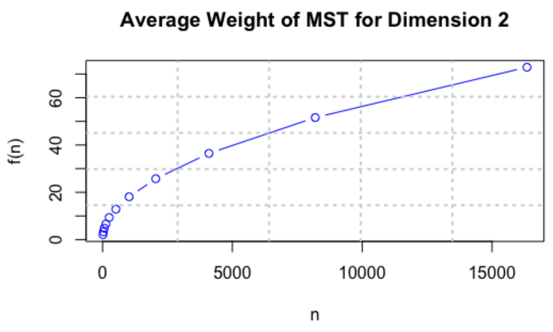

Figure 2: Average Weight of MST

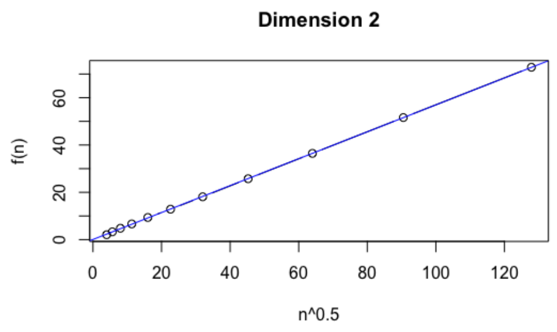

Figure 3: Fit to a function

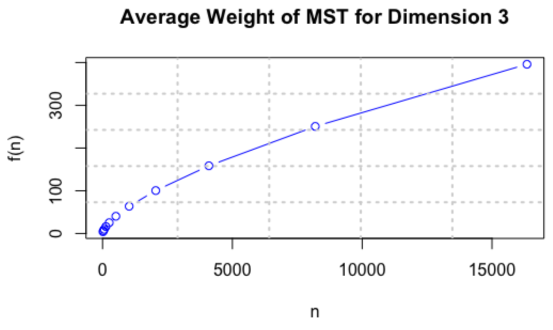

Figure 4: Average Weight of MST

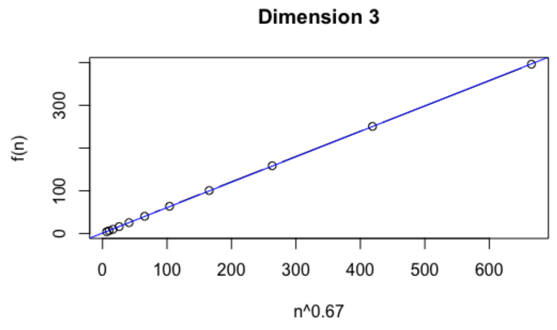

Figure 5: Fit to a function

When dimension = 3, the model is similar to the one used for dimension=2 from the graph. So we first also tried $n^{0.5}$, but it turned out that our model cannot fit the line very well, then we tried 0.6, 0.7, 0.8 in turn and found that the model with 0.7 performs best but $R^2 \neq 1$. We narrowed down 0.7 to 0.67 and got $R^2 = 1$. Therefore, the final model is $f(n) = 1.05139 + 0.59507n^{0.67}$, which is showed in figure 5.

When dimension = 4, the method is similar to the one used for dimension = 2. We first tried $n^{0.67}$, but it could not fit the line very well, so we tried 0.7, 0.75, 0.8 and found that the model with 0.75 performs very well and also got $R^2 = 1$. Therefore, the final model is

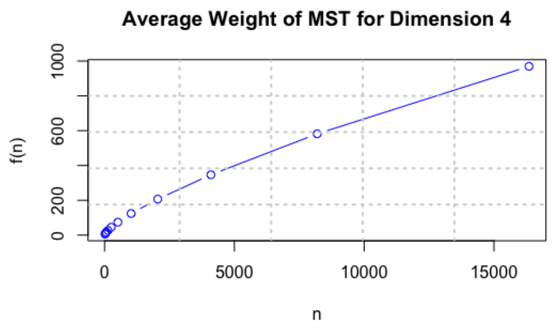

Figure 6: Average Weight of MST

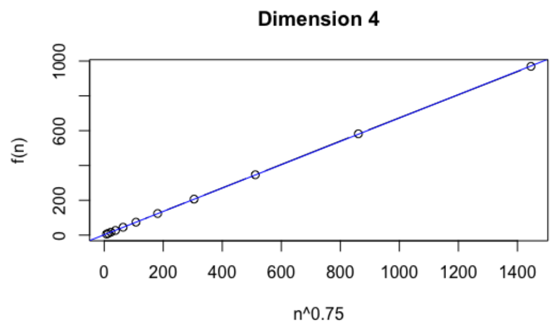

Figure 7: Fit to a function

$f(n) = 1.8768142 + 0.670696n^{0.75}$, showed in figure 7.

Second Part

Choice of Algorithm

We chose Kruskal's algorithm to implement generating MST over Prim's, because it has the same run time as Prim's, $O(ElogV)$, but it only requires tracking a list of all the edges in the tree as candidates, rather than updating the list of candidates on each iteration, which is easier to implement.

Analysis of Growth Rates

When dimension=0, the result is not very surprising. The expected weight of MST would be stable at around 1.2 with the increase in n. This makes sense, since the edge weights are all independent, increasing the number of points would also increase the number of very small weights. Since the number of edges in the graph grows faster than number of edges needed for a MST, while each larger MST contains more edges, it also contains a lower average edge cost. It seems that these two factors essentially cancel out, leave a constant MST weight.

When dimension = 2, 3, 4, the growth rates are not surprising too. First, we note that this function's second derivative is negative, indicating that greater values of n correspond to a less increase in $f(n)$. This makes sense, because increasing the number of vertices while limiting the coordinates of each vertex to a unit square/cube/hypercube results in vertices that are clustered more closely together, decreasing the impact of any additional vertices on total MST weight. Thus, a increase in n will not result in an equally proportional increase in $f(n)$, since new vertices are likely to be closer together to existing vertices. Second, we also

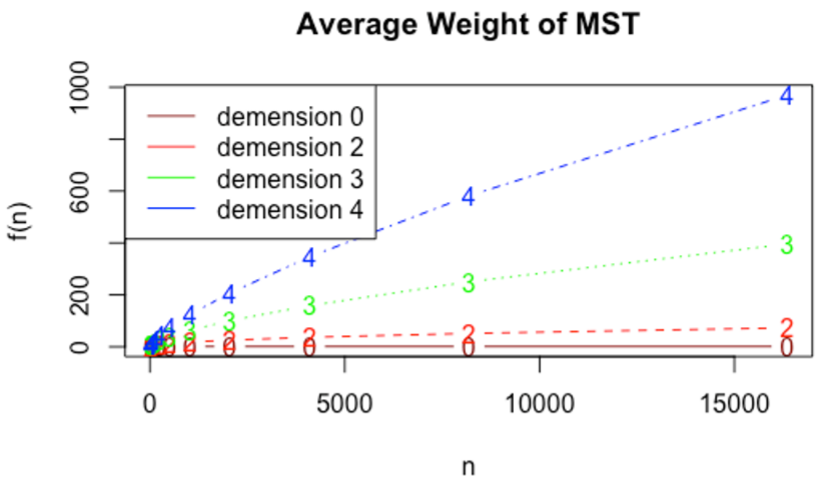

Figure 8: Average Weight of MST

note that as the numbers of dimensions increases, the growth rate of $f(n)$ approaches $O(n)$. Again, this makes sense, because increasing the number of dimensions that each vertex has access to decreases the clustering effect; that is, adding additional vertices are less likely to result in a graph that is tightly clustered, since these extra vertices are more likely to be farther away from existing vertices.

Time and Cache Size

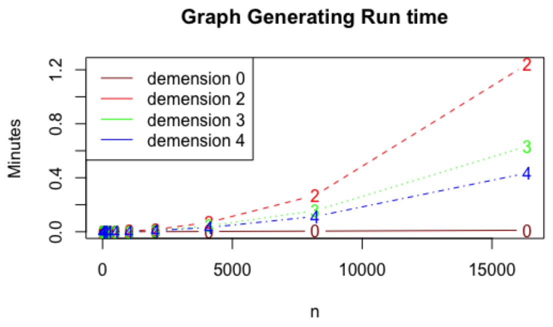

Figure 9: Generating Run Time

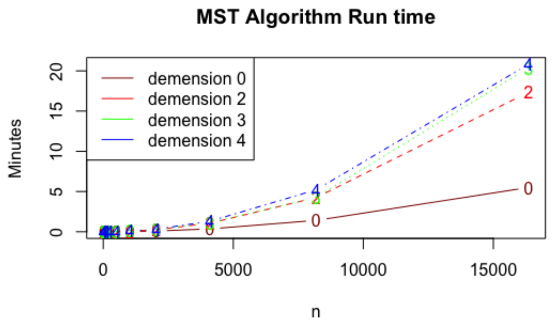

Figure 10: MST Run Time

The left graph is the run time for generating the graph, and the right graph is the run time for generating MST. As we expected, the run time for generating MST is increasing when dimension increases. This makes sense, because when dimension = 2, 3, 4, the calculation of

Euclidean distance has more terms. What is more, each doubling of n corresponded to about a 4x increasing in run time, since we include more edges in the MST. What is surprising is that generating a 2-dimension graph run slowest(left graph), maybe this is because our algorithm is not optimized enough for dimension 2.

Optimization
To optimize our problem, we need to optimize the edges we would handle with. For example, if we abandon the edges with large weight, the problem would run faster. Therefore, when dimension=0, we would abandon edges if the weight $\geq log_2(n)/n$ when n $>$ 512. When dimension =2, 3, 4, we abandon edges with weights $\geq 2/log_2(n)$, $\geq 3/log_2(n)$, $\geq 4/log_2(n)$ respectively.