ANLY550 Project1

Yi Li, Ziyan Liu

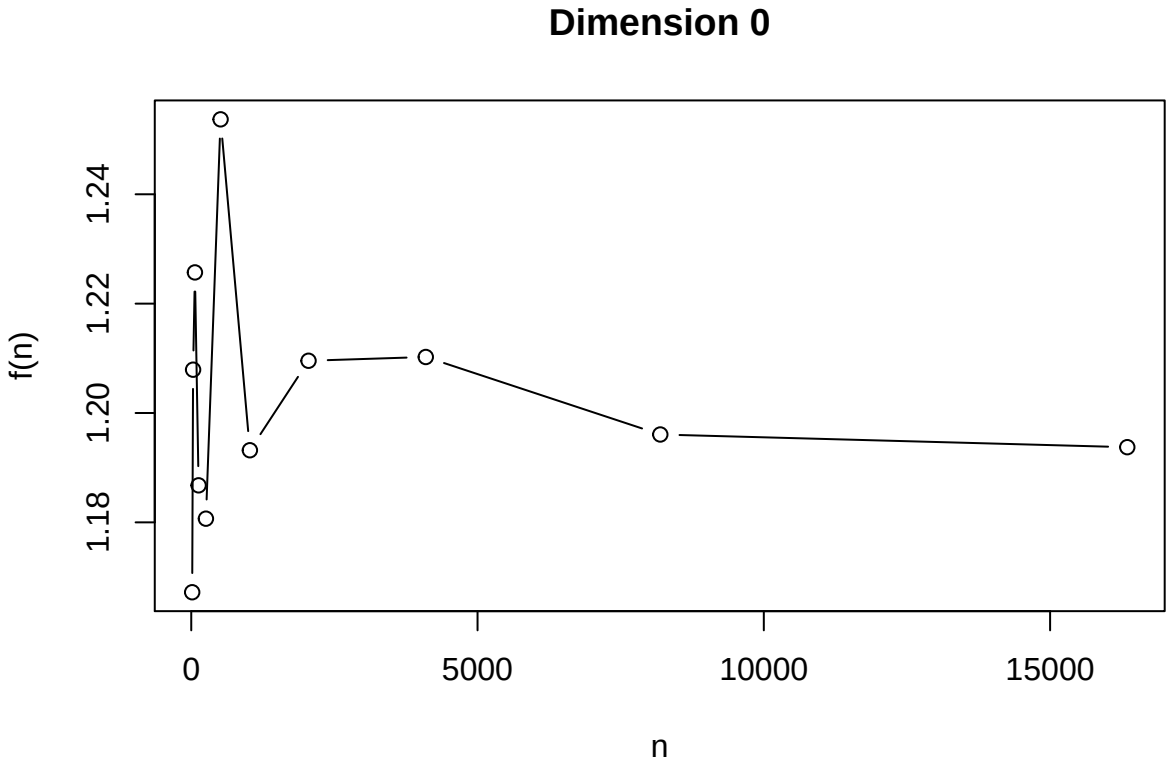
3/5/2017

The First Part of the Report

In this project, we implement the Kruskal's algorithm to generate the minimum spanning tree (MST). We use Rmarkdown to help us write this report, so you can see the R codes and R results directly.

Dimension = 0

```
data0 <- data.frame(matrix(c(
  16 , 1.1672175884029664 , 0 ,
  32 , 1.2079181268110901 , 0 ,
  64 , 1.2257097265748204 , 0 ,
  128 , 1.1867685637986477 , 0 ,
  256 , 1.1806702971405165 , 0 ,
  512 , 1.2537026183641438 , 0 ,
  1024 , 1.1931837956477647 , 0 ,
  2048 , 1.2095507211312981 , 0 ,
  4096 , 1.2102387130089962 , 0 ,
  8192 , 1.196066568913348 , 0 ,
  16348 , 1.1937450661426692 , 0
), byrow = T, ncol = 3))
plot(data0$X2 ~data0$X1, type = 'b', xlab = "n", ylab = "f(n)", main = "Dimension 0")
```



When n is small, the result is not stable. When n is large, we guess the function is $f(n) \approx 1.2$, which has nothing to do with n.

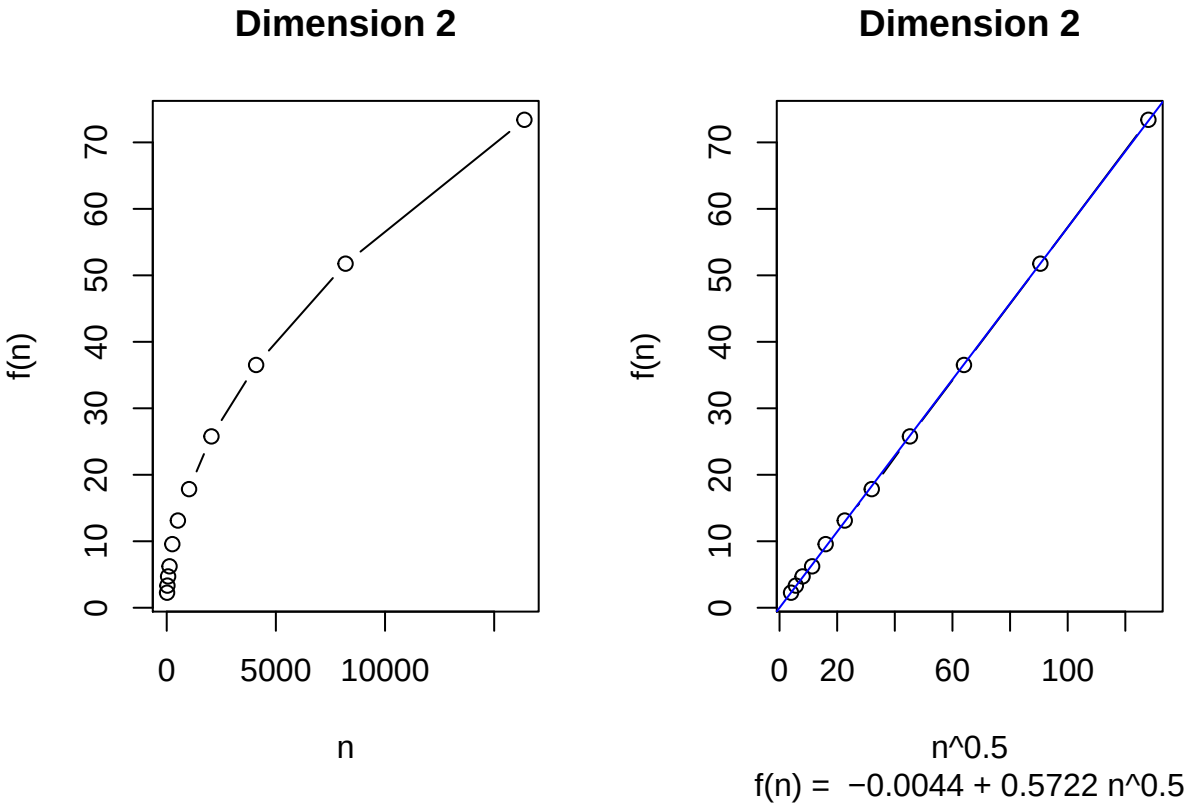
Dimension = 2

```
par(mfrow = c(1, 2))
data2 <- data.frame(matrix(c(
  16 , 2.2634525292386463 , 2 ,
  32 , 3.3271439476467557 , 2 ,
  64 , 4.709684116052539 , 2 ,
  128 , 6.228814676600714 , 2 ,
  256 , 9.567379745441391 , 2 ,
  512 , 13.115966864030796 , 2 ,
  1024 , 17.84070129748393 , 2 ,
  2048 , 25.773743834348107 , 2 ,
  4096 , 36.520109861811186 , 2 ,
  8192 , 51.758265571358315 , 2 ,
  16384, 73.39686387247293 , 2
), byrow = T, ncol = 3))
plot(data2$X2 ~ data2$X1, type = 'b', xlab = "n", ylab = "f(n)", main = "Dimension 2")

X1 <- (data2$X1)^0.5
plot(data2$X2 ~ X1, type = 'b', xlab = "n^0.5", ylab = "f(n)", main = "Dimension 2")
lm.fit2 <- lm(data2$X2 ~ X1)
summary(lm.fit2)
```

```
##
## Call:
## lm(formula = data2$X2 ~ X1)
```

```
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.46632 -0.10825 -0.02104  0.14586  0.41609
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.004436   0.106620  -0.042    0.968
## X1           0.572233   0.001954 292.857   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2483 on 9 degrees of freedom
## Multiple R-squared:  0.9999, Adjusted R-squared:  0.9999
## F-statistic: 8.576e+04 on 1 and 9 DF,  p-value: < 2.2e-16
```

```
abline(lm.fit2$coefficients[1], lm.fit2$coefficients[2], col=4)
title(sub = paste("f(n) = ", round(lm.fit2$coefficients[1],4), "+", round(lm.fit2$coefficients[2],4), "
```



We use regression to test our guess that $f(n) \sim n^{0.5}$, and get the function: $f(n) = \text{-}0.0044 + 0.5722\ n^{0.5}$.

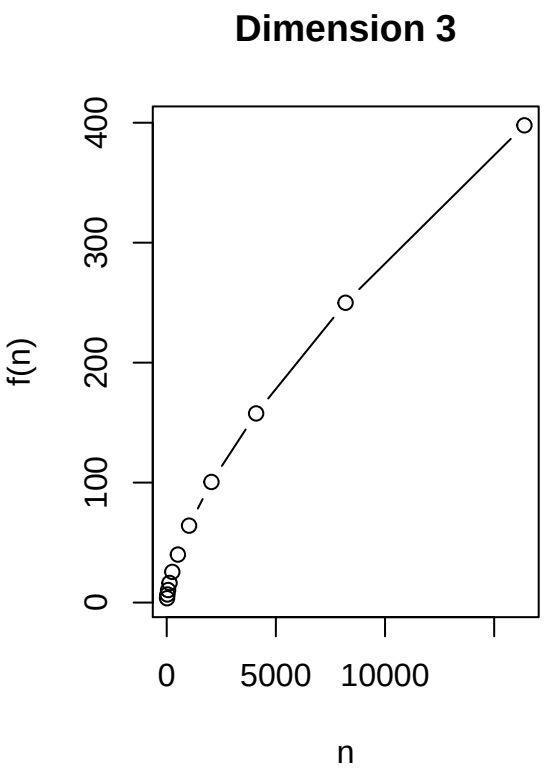
Dimension = 3

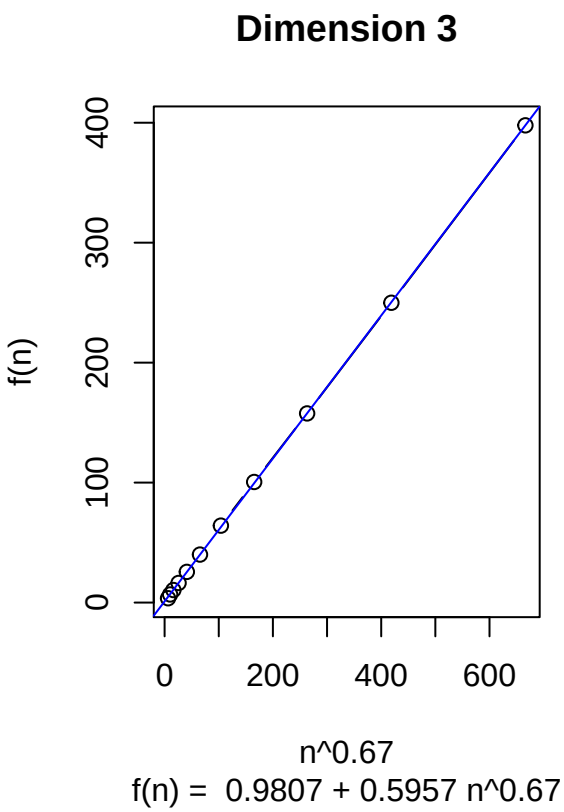
```
par(mfrow = c(1, 2))
data3 <- data.frame(matrix(c(
  16 , 3.598967422755377 , 3 ,
  32 , 6.756670403646506 , 3 ,
  64 , 10.431007308577867 , 3 ,
```

```
  128 , 16.31915988403701 , 3 ,
  256 , 25.566231152463722 , 3 ,
  512 , 40.00811589396948 , 3 ,
  1024 , 64.11373370185173 , 3 ,
  2048 , 100.52011976431889 , 3 ,
  4096 , 157.6689884444984 , 3 ,
  8192 , 249.92825384184837 , 3 ,
  16384 , 397.8033977865114 , 3
), byrow = T, ncol = 3))
plot(data3$X2 ~ data3$X1, type = 'b', xlab = "n", ylab = "f(n)", main = "Dimension 3")

X1 <- (data3$X1)^0.67
plot(data3$X2 ~ X1, type = 'b', xlab = "n^0.67", ylab = "f(n)", main = "Dimension 3")
lm.fit3 <- lm(data3$X2 ~ X1)
summary(lm.fit3)
```

```
##
## Call:
## lm(formula = data3$X2 ~ X1)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.19924 -0.25585 -0.07531  0.11166  1.20053
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.980721   0.268008   3.659  0.00524 **
## X1          0.595686   0.001038 574.060  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6921 on 9 degrees of freedom
## Multiple R-squared:      1,  Adjusted R-squared:      1
## F-statistic: 3.295e+05 on 1 and 9 DF,  p-value: < 2.2e-16
```

```
abline(lm.fit3$coefficients[1], lm.fit3$coefficients[2], col=4)
title(sub = paste("f(n) = ", round(lm.fit3$coefficients[1],4), "+", round(lm.fit3$coefficients[2],4), "
```





We use regression to test our guess that $f(n) \sim n^{0.67}$, and get the function: $f(n) = 0.9807 + 0.5957\ n^{0.67}$.

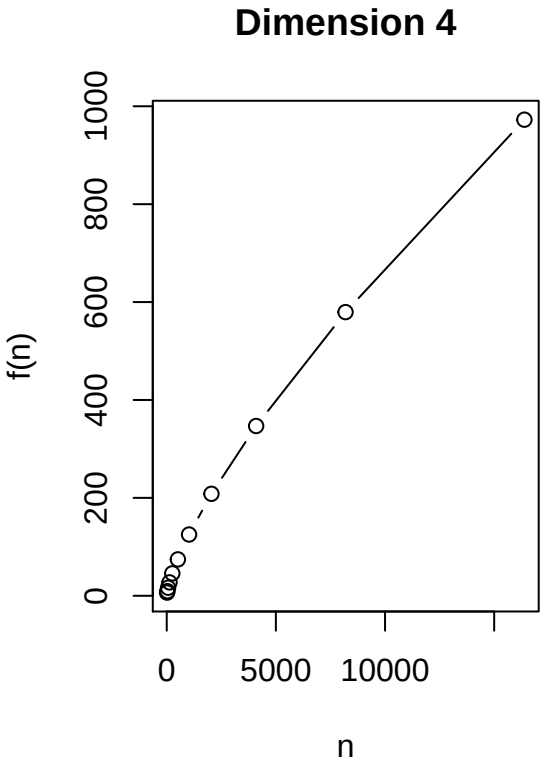
Dimension $= 4$

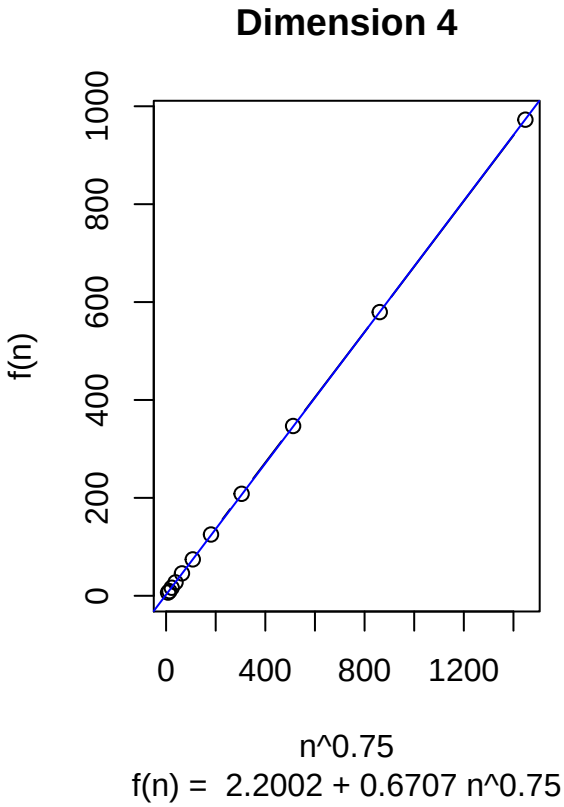
```
par(mfrow = c(1, 2))
data4 <- data.frame(matrix(c(
  16 , 6.528515542066423 , 4 ,
  32 , 9.421539495153944 , 4 ,
  64 , 16.443553523590857 , 4 ,
  128 , 27.35667731012516 , 4 ,
  256 , 45.73764069599367 , 4 ,
  512 , 74.30827006318503 , 4 ,
  1024 , 125.15709732233034 , 4 ,
  2048 , 208.3152192080025 , 4 ,
  4096 , 346.82834625961203 , 4 ,
  8192 , 579.5092089264452 , 4 ,
  16384 , 972.6004661388808 , 4
), byrow = T, ncol = 3))
plot(data4$X2~ data4$X1, type = 'b', xlab = "n", ylab = "f(n)", main = "Dimension 4")

X1 <- (data4$X1)^0.75
plot(data4$X2 ~ X1, type = 'b', xlab = "n^0.75", ylab = "f(n)", main = "Dimension 4")
lm.fit4 <- lm(data4$X2 ~ X1)
summary(lm.fit4)
```

```
##
## Call:
```

```
## lm(formula = data4$X2 ~ X1)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.8025 -0.9058 -0.2169  0.9209  1.9288
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 2.2001706  0.4709179   4.672  0.00117 **
## X1          0.6707011  0.0008672 773.453  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.254 on 9 degrees of freedom
## Multiple R-squared:      1,  Adjusted R-squared:      1
## F-statistic: 5.982e+05 on 1 and 9 DF,  p-value: < 2.2e-16
```

```
abline(lm.fit4$coefficients[1], lm.fit4$coefficients[2], col=4)
title(sub = paste("f(n) = ", round(lm.fit4$coefficients[1],4), "+", round(lm.fit4$coefficients[2],4), "
```

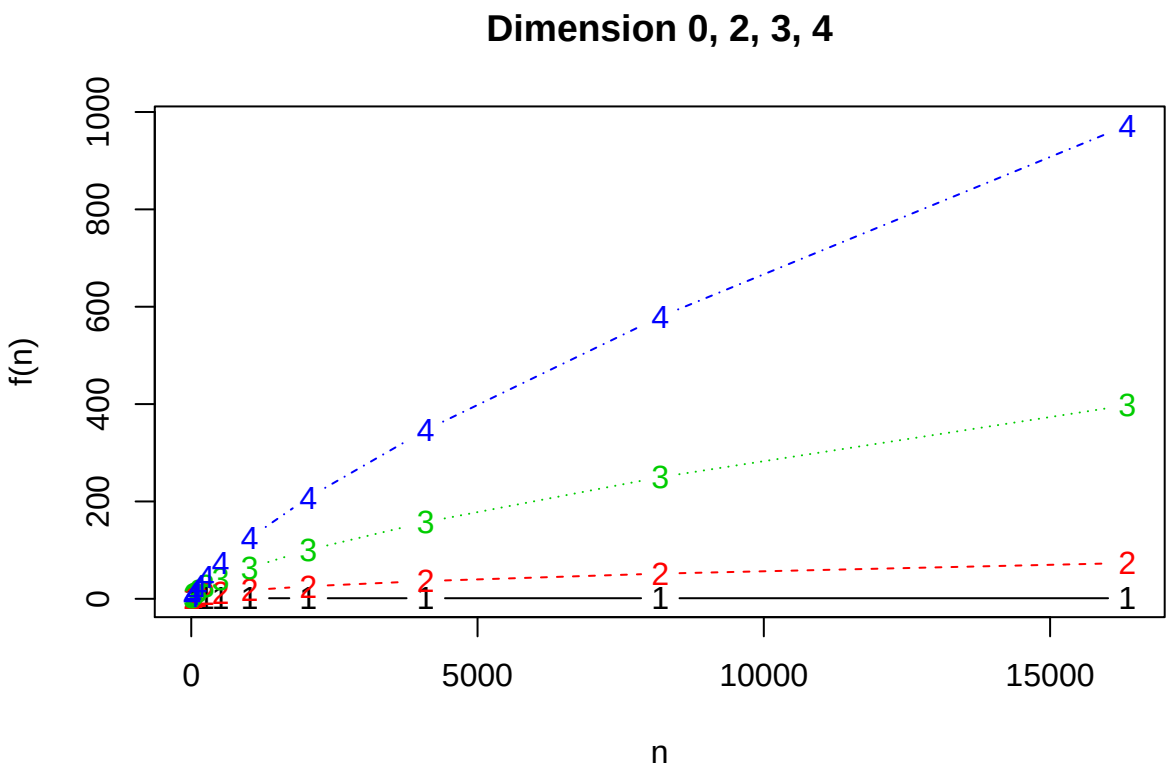




We use regression to test our guess that $f(n) \sim n^{0.75}$, and get the function: $f(n) = 2.2002 + 0.6707\ n^{0.75}$.

Graph of all dimensions

```
matplot(data0$X1, cbind(data0$X2, data2$X2, data3$X2, data4$X2), type = 'b',
        xlab = "n", ylab = "f(n)", main = "Dimension 0, 2, 3, 4")
title(sub = "Note: The black line represents the results of dimension 0.")
```



Note: The black line represents the results of dimension 0.

The Second Part of the Report

Analysis of the growth rates

When we generate a complete undirected graph with n vertices, we generate $\begin{pmatrix} n \\ 2 \end{pmatrix}$ edges, that is about $0.5n^2$ edges. To generate a MST, we only need $(n-1)$ edges. If we sort the edges, to avoid circles, we approximately need the samllest cn edges to generate a MST, where c is a constant value bigger than 1.

- For dimension = 0, the expected weight of the minimum spanning tree approximately maintains the value 1.2 when n becomes larger. We can explain this by considering the following example: When $n = 100$, we generate about 5000 edges. Suppose that we approximately need the smallest 120 edges to generate a MST. These 120 edges generated uniformly from [0,1] are samller than 0.024 in theory, and the average value of one edge is 0.012. So the weight of a MST is about $0.012 * 100 = 1.2$. This explains why the weight of the MST does not grow with the n. When n is small, the variation of the generated random numbers is bigger, so the result is not as stable as the result of big n. The result of large n keeps about 1.2 also let us believe that the random number generator is trustable.
- For dimension = 2, 3, 4, the weight of an edge is the Euclidean distance between its endpoints. When n becomes larger, there are more points in the unit (square, cube, or hypercube). Sine the points are denser, the expected weight of an edge becomes smaller. This explains why the growth rates are not linear and the plots of these functions are curves instead of lines. As the dimension increases, the space in the unit (square, cube, or hypercube) becomes larger (the longest edge is $\sqrt{2}, \sqrt{3}, \sqrt{4}$ respectively), so the weight of the MST also becomes larger.

Cache Size, Time, and Optimization

For a graph G with n vertices in the program, we set an edge $uv = vu$ and store each edge twice, so the size we need to store the edges is about n^2. As the n increases, the cache size becomes much larger, and the time to run the program becomes much longer. There are two parts of time: one is the time to generate the graph G, and the other is the time to generate the MST from the G. Both of them increase as the n increases. To save the cache size and improve the running time, we optimized the program by throwing away edges with large weights. And the optimization makes the program run much faster.

We optimized the program as the following:

- For dimension = 0, we throw away edges $\geq \log_2 n/n$ when $n > 512$;
- For dimension = 2, we throw away edges $\geq 2.0/\log_2 n$;
- For dimension = 3, we throw away edges $\geq 3.0/\log_2 n$;
- For dimension = 4, we throw away edges $\geq 4.0/\log_2 n$.

We checked the length of each generated MST to make sure that it is still $(n-1)$, so the optimized program gives the same results as the non-optimized program. For dimension = 0, we also set the threshold that do the optimization when $n > 512$, because when n is small, the results are not stable (sometimes the length of the generated tree is less than $n-1$). Our optimization method may not be the best way to save the size, but it does improve the program greatly and give the same results.