PROGRAMMING PSET 1 - CS 124

JUSTIN XIE AND KEVIN PATEL

1. Algorithm/Data Structure Selection

We considered two algorithms for finding the MST on a graph: Kruskals and Prims. We note that since the graph is complete and undirected, there are $\binom{n}{2}$ $= \frac{n(n-1)}{2}$ edges (equivalently, $\frac{|V|(|V|-1)}{2}$ edges). Functionally, the edges are growing exponentially relative to the number of vertices. For the remainder of this assignment, we will use $n = |V|, m = |E|$. We now consider the runtimes for Kruskals and Prims separately:

1.1. **Kruskals.** Before we can utilize the union-find data structure and begin building our MST, we must sort our edges (by weight). Assuming we use mergesort, our algorithm will require $O(m \log m)$ runtime to sort the edges.

Next, as proven in class, the amortized runtime of the algorithm is $O((m + n) \log^* n)$. Therefore, given that $m \approx n^2$, we have that the runtime for Kruskals including the sort is $O((m + n) \log^* n) + O(m \log m) = O(m \log m) = O(n^2 \log n^2)$.

1.2. **Prims.** Next, we consider Prims. We note that for Prims, the runtime is the same as Dijkstra. There will be m insert operations and n deletemin operations. Therefore, given that $m = n^2$, we want to choose the data structure that minimizes the runtime on insertion. We choose to use a binary heap because it has $O(logn)$ runtime for insertion and $O(logn)$ runtime on deletemin. Therefore, the runtime with a binary heap is $O(n \log n + m \log n) = O((n + m) \log n)$.

Ultimately, we choose to implement Prim's algorithm with a binary heap, which is faster on dense graphs where $\log n < \log m$. Although we ran into some issues (bugs) with implementation, our solid understanding of the algorithm and data structure allowed for us to address those issues, make optimizations, and ultimately get results for all n values with good running times. We felt that a Fibonacci heap would be too laborious to implement in a limited time, despite the more efficient runtime.

2. Results

We found that when running with 0 dimensions, the total cost would always be around 1.2 regardless of input size. Below is a chart showing our average costs when running with 0/2/3/4 dimensions by input size. We ran each dimension and number of vertices 5 times.

Date: February 24, 2021.

Vertices	0 Dimensions	2 Dimensions	3 Dimensions	4 Dimensions
128	1.217	7.54	17.722	28.443
256	1.182	10.732	27.899	47.491
512	1.21	14.781	43.395	78.675
1024	1.199	21.116	67.988	130.013
2048	1.192	29.541	106.463	216.493
4096	1.182	41.719	169.643	360.29
8192	1.193	58.903	266.971	603.449
16384	1.23	83.241	422.367	1009.08
32768	1.223	117.507	668.886	1688.817
65536	1.218	165.969	1058.441	2827.228
131072	1.194	234.442	1678.962	4739.13
262144	1.192	331.633	2657.174	7955.278

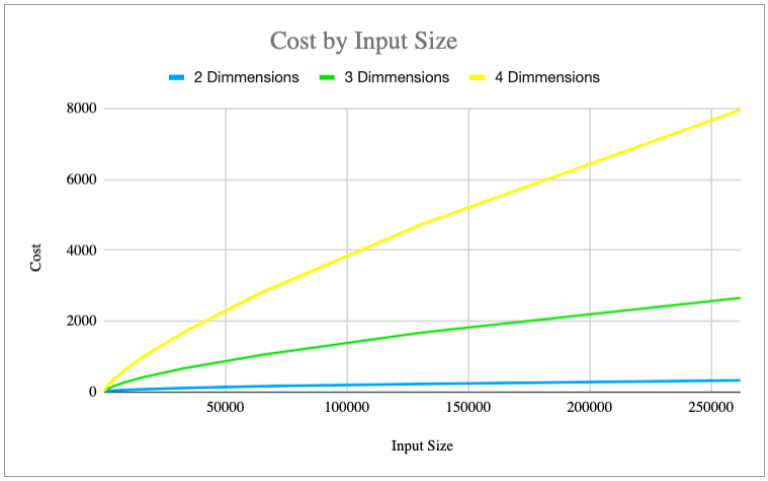

FIGURE 1. The cost/weight of the MST as a function of then umber of vertices. Values obtained from the table above. We do not illustrate dimension 0

.

Below we have the same results represented in terms of log_2. We believed that log_2 was a natural choice given the choice for the number of vertices (all powers of two). We illustrate the tables below.

log2(input)	0 Dimensions	2 Dimensions	3 Dimensions	4 Dimensions
7	.263	2.915	4.147	4.83
8	.251	3.424	4.802	5.57
9	.261	3.889	5.439	6.298
10	.26	4.4	6.087	7.023
11	.254	4.885	6.734	7.758
12	.251	5.383	7.407	8.493
13	.257	5.88	8.061	9.238
14	.261	6.38	8.722	9.979
15	.262	6.877	9.386	10.722
16	.261	7.375	10.048	11.465
17	.252	7.873	10.713	12.21
18	.255	8.373	11.376	12.958

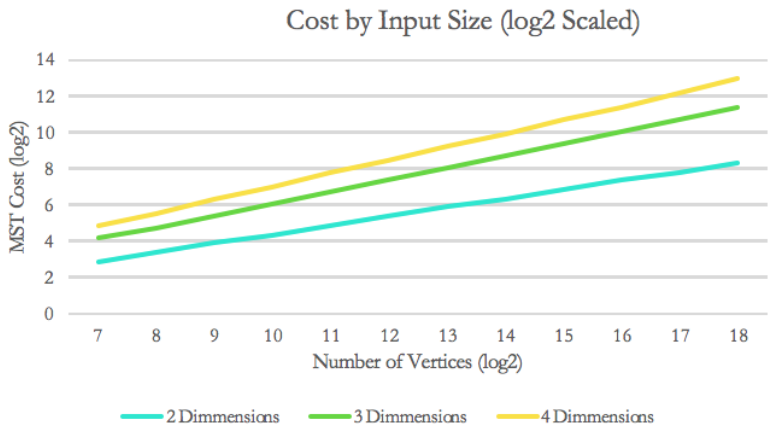

FIGURE 2. After taking $\log_2$ of the number of vertices and the associated cost, we have a linear relationship between the number of vertices and the MST cost.

3. Our prediction of $f(n)$ & our Experimental Results

3.1. **Dimension 0 Case.** Given our results, we first considered the case of $f(n)$ in the zero dimension case. In this case, we are randomly generating edge weights. We have that $f(n)$ for $n \in \{128, 256, \ldots, 262144\}$ are such that $f(n) \approx 1.2$ for all $f(n)$. The expectation of our edge weights e_i are such that for $\binom{n}{2}$ edge weights, if we were to enumerate them in sorted order, such that $e_i \leq e_j \forall j \in \{i+1, \ldots, \binom{n}{2}\}$, then $\mathbb{E}(e_i) = \frac{i}{\binom{n}{2}}$. We can then at least lowerbound the cost by taking the sum over

the smallest $n-1$ edges ($n-1$ edges are needed to make an MST):

$$\begin{aligned}\sum_{i=1}^{n-1} \frac{i}{\binom{n}{2}} &= \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} \\ &= \frac{2}{n(n-1)} \times \frac{n}{2}(n) \\ &= \frac{n+1}{n}\end{aligned}$$

The $\lim_{n\to\infty} \frac{n+1}{n}$ is 1. Therefore, we have a lower bound of 1 because we know that the smallest $n-1$ edges does not necessarily make an MST. Given that $f(n) \approx 1.2$ (in fact, $f(n)$ is within the range of 1.2 ± 0.05), we suspect that $f(n)$ for zero dimensions should actually be a constant, and asymptotic behavior $\lim_{n\to\infty} f(n) = C \approx 1.2$. We think C is likely the result of some converging infinite sum (or sum of some infinite series), but we do not have a strong guess as to the actual constant or series.

3.2. **Higher Dimensions.** Theoretically, since our points are randomly generated in a cube (or hyper cube), every coordinate is essentially a random point chosen uniformly from $[0, 1]$. Then, if we have dimension d and two uniforms $X_1 \sim U(0,1), X_2 \sim U(0,1)$, we can find the expected difference $\mathbb{E}(|X_1 - X_2|)$. By linearity of expectation, the expected value of edge weights are then

$$\sqrt{\sum_{i=1}^{d} \mathbb{E}(|X_1 - X_2|)} = \sqrt{d}\sqrt{\mathbb{E}(|X_1 - X_2|)}$$

Therefore, we expect the dimension to play a role in the constants of our solution $g(n, d)$.

With that in mind, a couple natural choices for $g(n, d)$ are $\sum_{i=0}^{j} a_i x^i$ (polynomial growth), $(1 + r)^n$ (exponential growth), $a_0 \log(n)$ (as suggested by the assignment handout). We took the $\log_2$ of our experimental results, which linearizes our vertices and our results. See figure 2. The approximate slopes are $0.5, 0.65$ and 0.74 for dimension $2, 3, 4$ respectively.

We have that the behavior is close to linear after the $\log_2$, so we expect $\log_2(g(n, d)) = \log_2(n^{c_d}) + \log_2(y_d) = c_d \log_2(n) + \log_2(y_d)$ where c_d, y_d are dependent on the number of dimensions. Then, we would expect $g(n, d) = y_d n^{c_d}$. From the slopes, we expect c_2, c_3, c_4 to be the result of some function c where $c(2) = 0.50$, $c(3) = 0.65$, $c(4) = 0.74$.

For higher dimensions, the slopes are $0.782, 0.8165, 0.8749, 0.9397, 0.9792$ for dimensions $5, 6, 10, 20, 100$ respectively. We noted that it appeared that $\lim_{d\to\infty} c_d = 1$. Looking at the growth rate, we hypothesize that the function should be $c(d) = \frac{d-1}{d}$. We plot $c(d) = \frac{d-1}{d}$ against our plot in the figure 3. This gives a least squares error of 0.00184. Therefore, we expect $g(n, d) = y_d n^{\frac{d-1}{d}}$ for $d > 1$. Now, to determine y_d, we note that $\log(y_d)$ is $-0.5352, -0.1796, -0.4201, -0.3021, -0.0998, 0.332, 0.6978, 0.9605, 1.6581, 1.9472$ for dimensions $2, 3, 4, 5, 6, 10, 20, 40, 70, 100$. Given these numbers, we hypothesize

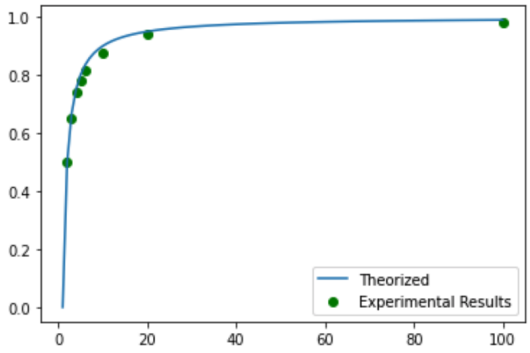

FIGURE 3. Our theorized exponent $c(d) = \frac{d-1}{d}$ against our experimental results for $c(d), d \in \{2, 3, 4, 5, 6, 10, 20, 100\}$. Least squares error of 0.00184.

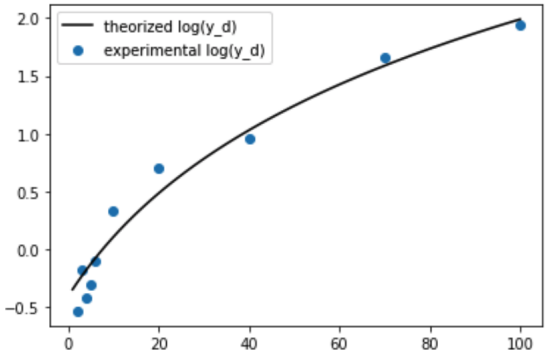

FIGURE 4. Our estimate on the constant y_d as a function of d. We have that $y_d = y(d) = 0.032 * x + 0.753$.

that $y(d) = 0.032 * d + 0.753$. See figure 4. Therefore, we have that:

$$g(n, d) = y(d)n^{c(d)}$$
$$y(d) = 0.032d + 0.753$$
$$c(d) = \frac{d-1}{d}$$

We threw out the possibility of an exponential growth solution by noticing that growth slowed as $n \to \infty$. We also attempted a polynomial fit (just as a sanity check that a polynomial solution was not correct). The polynomial interpolation is shown below. They were poor fits.

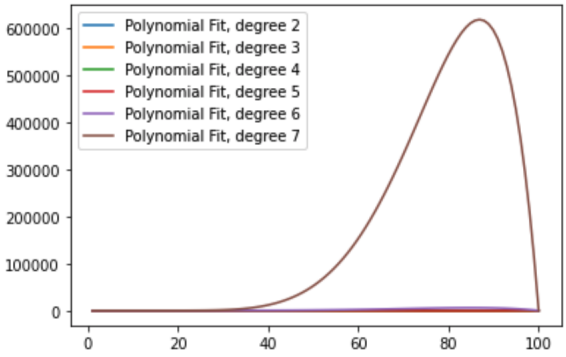

FIGURE 5. Polynomial fit of various degrees matched against our experimental c_d.

4. Optimization & Timing

We had to make many optimizations to successfully run the 2^{18} case. First, on memory:

(1) First, on the 2^{16} case, our computers stopped the program because the program was using too much memory. At the time, we were pre-computing all the edges and using an adjacency matrix to store the edges. In essence, we were using $n^2 \times 8$ bytes of memory, which comes out to $2^{32} \times 8 = 2^{35}$ bytes which is more than the 16 GB of memory. This hurdle forced us to calculate edge weights on the fly rather than pre-compute them.

Second, on timing. Initially, we could not reasonably expect to compute the $2^{18} - 262144$ case in any meaningful amount of time. It was taking us $> 40,000$ seconds (approximately 11 hours) to run one trial of $n = 262144$ for dimension 0. We needed to speed up our program considerably from a timing perspective to have any reasonable chance of running 5 trials of 2^{18} vertices. To optimize, we did the following:

(1) We realized that our function named "euclid", the function that calculates the distance between two vertices was our most expensive operation.In particular, we were using two standard library operations: *std::pow* and *std::sqrt*. We realized that $x * x$ was faster than $std :: pow$, with the time difference being very meaningful for large values n where *std::pow* might be called n^2 times. Since this is a complete graph and to calculate the distance between two vertices $a = (a_1, \ldots, a_d), b = (b_1, \ldots, b_d)$ for higher dimensions, we require computing $\sqrt{\sum_{i=1}^{d} std :: pow(b_i - a_i, 2)}$ on the order of n^2 times. We show the runtime differences in Figure 6.
(2) In addition, the *std::sqrt* was also a fairly expensive operation. We opted to only apply the square root when we pulled the minimum edge from the Heap, rather than for every edge we put into the heap. This would not change the minimum edge in the heap because edge weights are positive, so

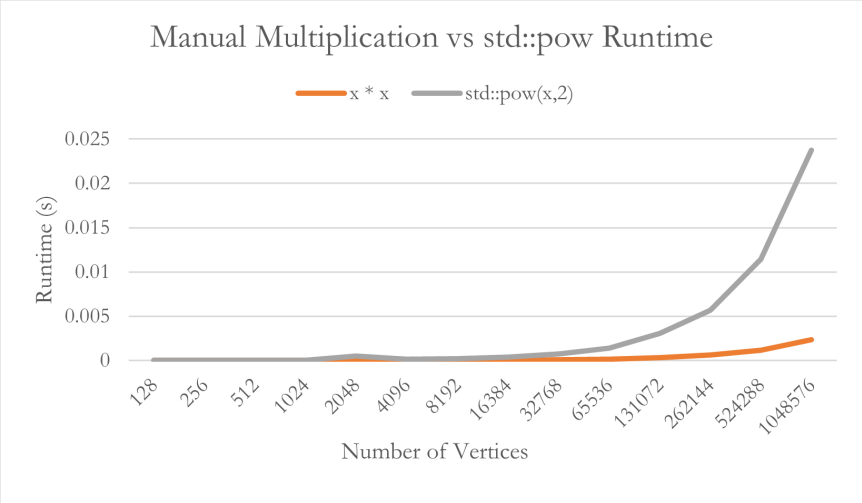

FIGURE 6. The runtime for manually multiplying $x * 2$ is orders of magnitude greater than that of *std::pow*.

$a^2 < b^2$ implies $a < b$. Rather than applying *std::sqrt* n^2 times, we now only apply the square root $n - 1$ times. .

(3) Third, a small but subtle optimization reduced our runtime by another $1/3$ and was particularly effective at trimming runtime for large values of n. In our implementation of Prim's, after retreiving a minimum edge to a vertex v not previously in the MST, we begin adding edges of the form $(v, w), w \in V$ into the tree. Rather than calculating distance immediately, we checked if the vertex had been previously visited already. Then, at any given step of Prims, if our MST has m vertices, our distance function would only be called $|V| - m$ times rather than $|V|$ times. Therefore, instead of $|V|^2$ calls to euclid, we reduced the number of calls to $\sum_{i=1}^{|V|} i = \frac{|V|^2+|V|}{2}$. For larger values of $|V|$, $\frac{|V|^2+|V|}{2} \approx \frac{|V|^2}{2}$, reducing the total number of calls to euclid by $\frac{1}{2}$, a practically meaningful improvement. See figure 7 and 8 for exact implementation changes.

```
double MST_sum = 0;
while(!H.isEmpty()){
    pair<int,double> v = H.deletemin();
    MST_sum += v.second;
    visited[v.first] = 1;
    for(int i = 0 ; i < vertices; i ++){
        double currDist = euclid(vertex_locations[v.first],vertex_locations[i]);
        if(i != v.first && visited[i] == 0 && currDist < dist[i]){
            dist[i] = currDist;
            H.insert(i,dist[i]);
        }
    }
}
return MST_sum;
```

FIGURE 7. The Pre-Optimized code. Our euclid function is called $|V|$ times after every deletemin from the heap.

```
for(int i = 0 ; i < vertices; i ++){
    if(visited[i] == 0 && i != v.first){
        double currDist = euclid(vertex_locations[v.first],vertex_locations[i]);
        if(currDist < dist[i]) {
            dist[i] = currDist;
            H.insert(i,dist[i]);
        }
    }
}
```

FIGURE 8. The Post-Optimized Code. In comparison to Figure 7, the post-optimization code only calls our "euclid" function if the vertice has not already been visited. Therefore, if our MST has m nodes, we only call euclid $|V| - m$ times rather than m times. Practically speaking, this drastically improved runtime even though the big-O runtime is the same.

Altogether, these small optimizations improved our runtime by nearly 2/3, cutting down our runtime in the $n = 262144$ case from $> 40,000$ seconds to $< 15,000$ seconds, a much more manageable runtime of four hours per trial per dimension. We show our runtimes below.

4.1. **Exact runtimes.** We can see that each time we double our input size, the total time to compute is roughly quadrupled. This is faster than we expected because our running time is approximately $O(|V|^2 \log(|V|))$; however, it behaves as if its running in $O(|V|^2)$. It's also clear that the increases in number of dimensions results in an increase in time. However, not nearly as much as increasing the input size. This is largely because we only go from 0-4 dimensions. If we were to run with many more dimensions, we would see much longer run times as euclid() and generatedim() run in $O(|V| * dimensions)$ or $O(|E| * dimensions)$ time.

Input Size	0 Dimensions	2 Dimensions	3 Dimensions	4 Dimensions
128	0.003	0.004	0.004	0.004
256	0.015	0.015	0.015	0.015
512	0.057	0.053	0.06	0.06
1024	0.222	0.222	0.325	0.257
2048	0.861	0.847	0.902	0.917
4096	3.595	3.544	3.604	3.91
8192	13.886	14.327	14.213	14.413
16384	55.370	56.576	57.775	61.081
32768	219.716	221.631	229.174	234.352
65536	864.759	866.663	934.832	938.88
131072	3355.466	3459.91	3658.911	3761.683
262144	13427.896	14062.179	14461.919	14770.689