Zhen, Zhong

zz197@georgetown.edu

Programming Assignment 1 Write-Up

Analysis of Algorithm

1. Adjacency List Representation

We will implemented an adjacency lists as a vector of lists. We will consider two cases: (1) dimension=0. (2) dimension> 0. We initiated graph and stored s adjacency lists. The tuple of nodes and weight will store in list with index if there is a directed edges. We build an array of n lists, one for each vertex in the set of nodes. Each list contains all the vertices that are adjacent to one node. Since the graph is undirected, so each edge will appears twice in the lists of both nodes and follow the symmetrical pattern.

We firstly need to make a selection of graph representation between adjacency list and adjacency matrix. Although adjacency matrix gain an advantage in time-saving and handling the dense graph over adjacency list, adjacency list only requires $\theta(V + E)$ memory while adjacency matrix requires $\theta(V^2)$ memory which saves great memory and avoid the memory-ran-out since the number of vertices is large. Moreover, adjacency list can quickly list all the vertices that is adjacent to a vertex which is superior to adjacency matrix. Therefore, we choose adjacency list to represent our graph after making trade-off between time and memory.

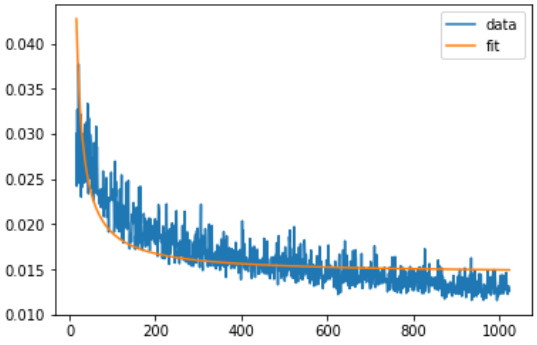
2. K(n) Selection

Before we set up the k (n), we quickly ran out of memory starting at n value of 8192, so we need to set up the k (n) as the limitation of edge of weight to drop the edges of weight larger than k (n) when we increase the n size. It not only decreases the running time for the larger n but also save more memory to run the program.

We will execute average maximum weight for each smaller n vertices and fixed number of trails of 20 in different dimension by dividing the total maximum weight by the number of trails. The table listed on the left is the table of average maximum weight for different n. We can find out for deimension=4, the average maximum weight is largest and conclude that the k(n) of average largest weight for dimension=4 is also largest.

Table Of Maximum Weight for Different n (number of trail=20)

n	d=0	d=2	d=3	d=4
16	0.20	0.36	0.49	0.62
32	0.12	0.27	0.41	0.53
64	0.07	0.19	0.34	0.46
128	0.04	0.14	0.27	0.40
256	0.02	0.11	0.22	0.34
512	0.013	0.08	0.18	0.29
1024	0.0070	0.06	0.15	0.25
2048	0.0041	0.04	0.12	0.22
4096	0.0023	0.02	0.1	0.21



We use the average maximum weight calculated by continuous n from 16 to 1024 to estimate k(n). The graph listed on the right indicates that the blue line represents the plot of exact average maximum weight of n from 16 to 1024. We can find out that blue line follows as the inverse proportion function and has the format of $k(n) = \frac{a}{n} + b$. Then we can estimate the fitted curve of the blue line as $k(n) = \frac{0.5}{n} + 0.01$. Therefore, we can make a scale and set the $k(n) = \frac{60}{n}$ and assume that n will range from 16 to 32768. This k(n) means that when the weight of an edge is larger than $k(n) = \frac{60}{n}$,we will cut off this edge in order to save more memory and time. Howver, this k(n) only influences the run-time, but not the results of the average minimum spanning tree weight.

3. Prim Algorithm Implementation

We use a boolean array used[ ] to represent whether the set of vertices included in MST. If a value used[index] is true, then vertex v is included in MST. Array dis[ ] is used to store key values of all vertices and we set $dis[0] = 0$ as first picked vertex. We set vertex as indexes of parent nodes in MST and initialize the $vertex = -1$ as root . We pick the minimum distance vertex from the set of vertices which is not yet processed and put the minimum distance vertex in the shortest path tree. We keep updating distance value of the adjacent vertices of picked vertex only if the current distance is greater than new distance and the vertex in not in the shotest path tree. For each execution, the weight will be accumulated and finally return to the total minimum spanning tree weight of MST.

4. Number of Vertices, Number of Trails and Dimension Implemtion

We classfied the problems into 2 cases: (1) dimension=0. (2) dimension>0. And we get the value of total minimum spanning tree weight by inputing different dimension. And we can get the average minimum spanning tree weight by dividing by the the number of trails.

Result

1. Result of Average MST Weight by different Number of Vertex and Dimension. (Fixed Number of Trails=5)

n	d=0	d=2	d=3	d=4
16	1.0472	2.8694	4.5946	6.3648
32	1.0395	3.9033	7.2110	10.0517
64	1.0514	5.3205	11.4176	17.0121
128	1.2169	7.7236	17.1301	28.5719
256	1.2700	10.6912	28.0740	46.8966
512	1.2117	14.8862	43.2993	77.9676
1024	1.2044	21.0189	67.8883	130.1686
2048	1.1829	29.7149	107.1594	215.7905
4096	1.2204	41.8673	169.7207	362.1176
8192	1.2117	59.0622	267.5354	603.3328
16384	1.1972	83.2122	422.8192	1010.0104
32768	1.2062	117.4884	669.1434	1688.9686

2. Result of Average MST Weight for different Number of Trails and Dimension. (For n =16, 32,64, 128, 256, 512)

n	trails	d=0	d=2	d=3	d=4
16	5	1.0472	2.8694	4.5946	6.3648
16	100	1.1411	2.6991	4.5379	6.1499
16	1000	1.1618	2.7068	4.5432	6.1457
32	5	1.0395	3.9033	7.2110	10.0517
32	100	1.1703	3.9316	7.1611	10.3934
32	1000	1.1926	3.8587	7.1522	10.3009
64	5	1.0514	5.3205	11.4176	17.0121
64	1000	1.2039	5.4430	11.2513	17.1678
128	5	1.2169	7.7236	17.1301	28.5719
128	500	1.2021	7.6160	17.6432	28.4288
256	5	1.2700	10.6912	28.0740	46.8966
256	500	1.1943	10.6730	27.6212	47.1429
512	5	1.2117	14.8862	43.2993	77.9676
512	200	1.2014	14.9542	43.3044	78.3082

3. Results of Process Time for different Number of Vertices, Number of Trails and Dimension

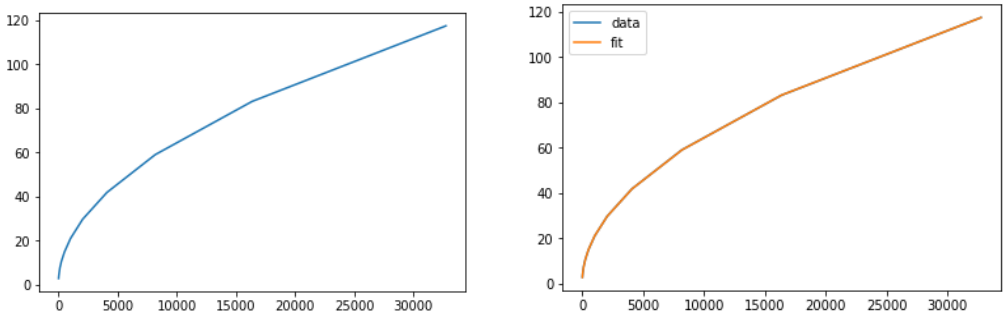
n	trails	sec(d=0)	sec(d=2)	sec(d=3)	sec(d=4)
16	5	0.0010	0.0021	0.0022	0.0026
16	100	0.0141	0.0359	0.0459	0.0566
16	1000	0.1176	0.4023	0.4413	0.5059
32	5	0.0026	0.0069	0.0083	0.0094
32	100	0.0414	0.1368	0.1527	0.1859
32	1000	0.4983	1.3902	1.5729	1.8082
64	5	0.0109	0.0273	0.0314	0.0340
64	1000	1.8229	5.1451	5.6652	6.7384
128	5	0.0278	0.0973	0.1149	0.1357
128	500	2.4524	9.9085	11.3364	15.7987
256	5	0.0824	0.3796	0.4096	0.4903
256	500	7.1443	35.9285	46.1604	49.7968
512	5	0.2692	1.6316	1.7314	2.1637
512	200	11.4592	56.4004	65.3202	75.8431
1024	5	1.2080	6.7248	8.3641	9.8098
2048	5	3.7994	23.4102	29.3048	33.4968
4096	5	11.9466	89.5362	101.3141	117.4496
8192	5	47.2135	287.8538	374.2124	458.8455
16384	5	181.8379	1305.7526	1551.6811	1839.5688
32768	5	1133.1752	4887.5388	4967.4611	6844.4575

Note that the process time shown above is total running time to run m trials in seconds. We can get the time per trials by dividing total running time by number of trails.

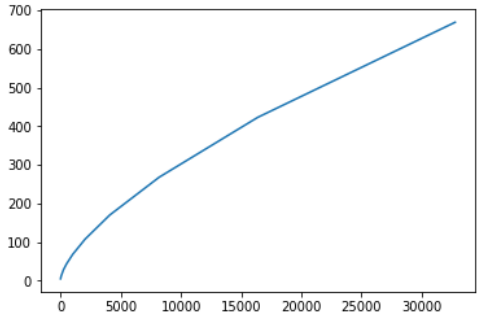
4. Estimation of f (n) by different dimension

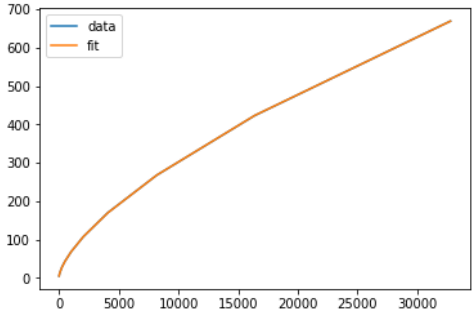
(1) For dimension=0, we can see that average MST weight always approaches to 1.20. Therefore, we can estimate f(n) for dimension 0 as $f(n) = 1.20$.

(2) For dimension=2, we can find relationship between n and average MST weight by plotting the graph. Based on the graph on the left, we can predict that f(n) follow the format that $f(n) = an^b + c$. Therefore, we can estimate the fitted curve that $f(n) = 0.66n^{\frac{1}{2}} + 0.18$. The graph on the right also indicates that $f(n) = 0.66n^{\frac{1}{2}} + 0.18$ fits the exact curve very well by plotting the estimated f(n) and real data point together.

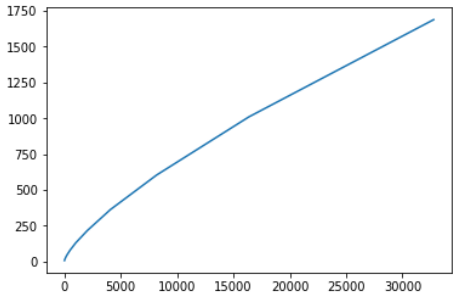


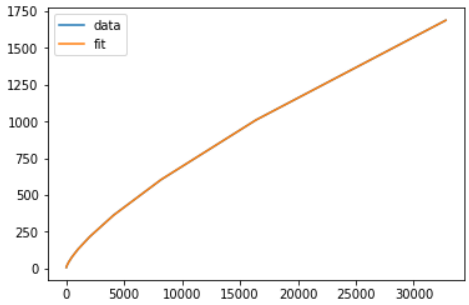
(3) For dimension=3, based on the graph on the left, we can estimate that f(n) has the format that $f(n) = an^b + c$. Therefore, we can estimate the fitted curve that $f(n) = 0.69n^{\frac{2}{3}} + 0.51$. The graph on the right also indicates that $f(n) = 0.69n^{\frac{2}{3}} + 0.51$ fits the exact curve very well by plotting the estimated f(n) and real data point together.





(4) For dimension=4, based on the graph on the left, we can estimate that f(n) has the format that $f(n) = an^b + c$. Therefore, we can estimate the fitted curve that $f(n) = 0.75n^{\frac{3}{4}} + 0.69$. The graph on the right also indicates that $f(n) = 0.75n^{\frac{3}{4}} + 0.69$ fits the exact curve very well by plotting the estimated f(n) and real data point together.





Discussion

1. Algorithm Selection and Comparison

In Minimum Spanning Tree, we usually have two popular algorithm: Prim algorithm and Kruskal's algorithm. Kruskal's algorithm relies on edges. It basically focuses on finding a subset of the edges which can form a tree including every vertex and achieved that total weight of all the edges in the tree is minimized. However, Prim algorithm relies on nodes. Its basic idea is to build a tree one vertex at a time from an arbitrary starting vertex and add a connection with lowest weigh from the tree to another vertex. In summary, Prim algorithm has better performance in dense graph while Kruskal's algorithm does better job in sparse graph. Moreover, it takes Prim's Algorithm $O(E \lg V)$ since we use adjacency list to represent the graph while it takes $O(E \lg E + E \lg V)$ for Kruskal's algorithm.

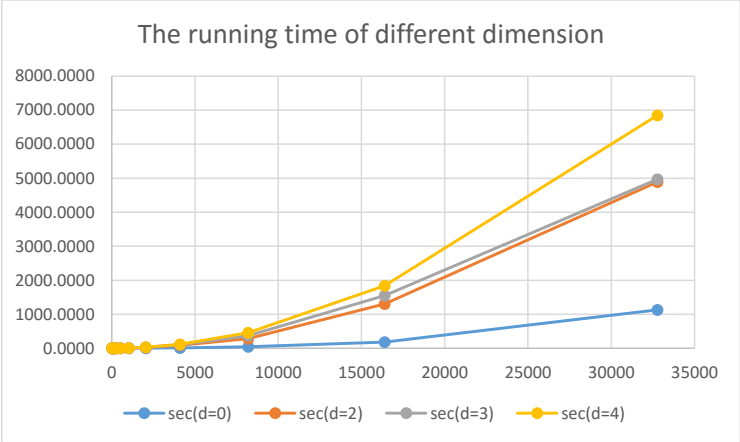
In this case, we use a completed graph and the vertices number n ranges from 16 to 32768, which have large number of nodes and form a dense graph. And it takes Therefore, Prim's algorithm is better choice for this case since it has better performance in dense graph.

2. f(n) Interpretation

The growth rates of f(n) is surprising to me that dimension is equal to denominator of the power in f (n). For example, when dimension is 2, the denominator of power in $f(n) = 0.66n^{\frac{1}{2}} + 0.18$ is also 2. It also has same pattern with higher dimension. I think it's because when we increase the dimension, there is also additional edges adding to original edges which leads to higher average minimum spanning tree weight.

3. Running time and Cache Size

It's clearly that running time increases exponentially as n increases for the same dimension, since the time complexity of Prim's algorithm is $O(E \lg V)$. We also find out that higher dimension will lead to more running time with same number of vertices and growth rate of high dimension increases faster than growth rate of the low dimension , which is shown by the graph below. When we construct a graph $G(V, E)$ with $O(n)$ edges, if we examine the $C1$-neighborhoods of the input points and dimension d to construct E, then the number of subcubes examined is $(2C_1 + 1)^d$, which takes $O(n(2C_1 + 1)^d)$ time to construct G. For the Euclidean minimum spanning forest problem of on G. In the repeated loop, it adds additional number of edges of $O(n\beta(2C_2 \lg n + 1)^d)$ to original edges (β is a constant that $\beta > 0$). Therefore,the run time of the Prim's algorithm will be $O(n(2C_1 + 1)^d + n\beta(2C_2 \lg n + 1)^d))$,which will greatly increases when d is increasing.



Cache size is highly related to our running time and largest number of vertices that we can run. We found out that when we run a large number of n, we will run out of the memory. Therefore, we need to set up a limitation in order to cut off some weights so that we can avoid memory run-out and save more time, such as setting up a k(n). Moreover, we can also use a computer with more memory to increase the speed of running.

4. Random Number Generator

Random Number Generator can generate a number of random graphs and helps us to decrease the bias of output. Random Number Generator generates a list of random number based on the seed. However, we need to note that there is no completely randomness. We can just approximate the randomness by increasing the number of trails.