CS124 Programming assignment 1 - Writeup-

Nibrass Fathi, Saketh Mynampati

February 2023

1 Problem Overview

We sought the average total weight of a minimum spanning tree of a complete graph with n vertices with various, as follows:

1) n vertices, where each edge has weight sampled from distribution $Unif(0, 1)$.

2) n vertices, where each vertex is placed randomly on a unit square, and the edge weight between two vertices is the Euclidean distance between them.

3) Same as (2), but in a 3D unit cube.

4) Same as (2), but in a 4D hyper cube.

From the graphs, we sought to find how increasing n affects this average weight of a spanning tree on the graph.

2 Quantitative Results

Note: 5 trials were run for each entry to create the average weight.

$f(n)$ for 0D graph	
$f(n)$	n
1.2751134	128
1.2913378	256
1.1934954	512
1.1747834	1024
1.1952761	2048
1.2071271	4096
1.2022366	8192
1.1966430	16384
1.1985922	32768
1.2072237	65536
1.2034093	131072

$f(n)$ for 2D graph on unit square	
$f(n)$	n
7.7734570	128
10.6628588	256
14.8853457	512
21.1870854	1024
29.4828830	2048
41.7430057	4096
58.8945534	8192
83.1593967	16384
117.4599722	32768
165.9432077	65536
234.2448664	131072

$f(n)$ for 3D graph on unit cube	
$f(n)$	n
17.7770464	128
27.7178233	256
43.6572031	512
68.2900971	1024
107.4693336	2048
169.6212335	4096
267.7217554	8192
422.1694711	16384
669.6672612	32768
1059.6590194	65536

$f(n)$ for 4D graph on unit hypercube	
$f(n)$	n
28.5471025	128
47.3687534	256
78.5385088	512
130.6891367	1024
217.2330969	2048
361.3054684	4096
604.7008975	8192
1009.1427692	16384
1691.2949036	32768
2825.6088969	65536

Note: We weren't able to run the algorithm for $n > 131072$, due to a loack of memory optimization.

3 Discussion of Results

We used the above tabulated results for each graph as input into regression, where we plotted various potential relationships between n and $f(n)$.

We break down the relationship on a case by case basis, and estimate f(n) using Desmos for regression.

1) This is merely $f = 0.2$, because

2) This grew approximately linearly with $\log n$, as evidenced by the linear incrementation of f as n doubled.

3) This grew quadratically.

4) This grew cubically.

These results are surprisingly orderly, because each grew with an exponent of *ndim* - 1.

4 Approach

4.1 Mersenne Twister (Note on Random Numbers)

Upon inspecting the C++ rand() function for documentation, it became apparent that it was limited in its ability to produce larger random values, due to two issues: low randomness of lower order bits, and lower RAND_MAX values. Thus, we decided to use the newer library. Not much extra consideration was given to this, but reliable randomness increases the quality of the results, so we thought it would be worth it.

4.2 Asymptotic Runtime

Our algorithm implements Prim's algorithm directly, with a select few optimizations. Creating the adjacency list is theoretically $O(n^2)$ due to nested for loops, but in practice, is much faster than Prim's algorithm because of the lower constant time (frequently just negligible) for each operation/comparison.

In the future, we should optimize this by mathematically finding the distribution from which we should sample edge weights, instead of generating them and discarding them.

Prim's, in theory, is $O(m \log n)$ but due to large binary heap inserts, it can also reach $O(m \log^2 n$ due to insertion into a bin heap being $O(\log n)$.

By pruning, we decrease m, but it is still $O(n^2)$, barring additional mathematical verification.

The general $O(n^2) \log^2 n$ was expected, but memory issues made it outperform this for smaller values, and for values about 100000, out algorithm was slower.

4.3 Algorithm

We combined Prim's with an edge pruning strategy as described below.

Adjacency List Representation

We needed to represent our graph in some way. We initially didn't explicitly create the graph, and calculated the validity and weight of edges as we iterated through Prim's, but this began to become very time intensive with larger heap inserts, so we decided to offload this into an Adjacency List.

The Adjacency list is better than the matrix, as it can handle sparse graphs; see **Handling large n.**

Prim's Algorithm and Binary Heap

As seen in class, Prim's algorithm is useful for finding the minimum spanning tree of a weighted undirected graph. In this algorithm, we start with an arbitrary node and gradually grow the minimum spanning tree by adding the edge with the minimum weight that connects the tree to a node that is not yet in the tree.

To efficiently find the minimum-weight edge to add at each step of the algorithm, Prim's algorithm uses a data structure called a priority queue, which is implemented using a min-heap. A min-heap is a binary tree data structure where each node is smaller than its children, and the root node is the minimum element in the heap.

By using a min-heap as the priority queue, we can efficiently extract the edge with the minimum weight in logarithmic time.

4.4 Handling large n

As per the hint, in order to handle large n, we wanted to consider simplifying the graph.
The hint says the minimum spanning tree is extremely unlikely to use any edge of weight greater than $k(n)$, for some function $k(n)$. We can estimate $k(n)$ using small values of n, and then try to throw away edges of weight larger than $k(n)$ as we increase the input size. The hint claims that throwing away edges in this manner never lead to a situation where the program returns the wrong tree.
Let's prove that this claim is true before using it in our code.

Claim: In some Minimum Spanning Tree of n vertices, for some function $k(n)$, and for some small integer n, throwing away edges of weight larger than $k(n)$ will not result in a wrong Minimum Spanning Tree.
Proof: We'll use proof by contradiction to prove this claim.
Assume for some function $k(n)$, and for some integer n, throwing away edges of weight larger than some $k(n)$ will result in Prim's Algorithm creating a Minimum Spanning Tree, T', that is not a valid MST of the original graph.

(Note: for $k(n)$ too small, Prim's may not be able to create a tree due to a disconnected graph. We are stating that if we are able to get a valid tree T', it is a valid MST of the original graph.)

We now have a tree T' with edges E. By definition, T' contains all vertices needed to be a valid MST for the larger graph. Assume there is some correct MST T'' that has total weight less than our T'. This means that every cut that T'' crosses, our T' also crosses, but there is some cut for which T'' has a smaller crossing edge than T'. However, this would mean our modified graph didn't contain that edge in the first place (Otherwise, our oracle to Prim's would have found it), which means it had length greater than $k(n)$. The corresponding edge in our T' is, by definition, less than this $k(n)$, so we have a contradiction.

We ran the complete graph on smaller values to use regression (on Desmos) to determine an optimal $k(n)$:

```
npoints > 4000 ? pow(10, (1.0/ndim)) * pow(npoints, -(1.0/ndim)) : 2
```

5 Walkthrough of code setup

We started by implementing a function that would generate a random graph by a given size and dimensionality, with random edge weights or points in a certain space. It then proceeds to implement Prim's algorithm, which is a greedy algorithm used to find a minimum spanning tree for a weighted undirected graph. The minimum spanning tree is represented as a vector of tuples that contain information about the edges included in the tree.

The input parameters to the code are passed through the command line and are as follows:

- flag:
  a integer that determines which type of graph to generate (0 is empty, 1 is random edge weights, 2 is unit square, 3 is unit hypercube, 4 and 5 are testing)
- npoints:
  the number of vertices in the graph
- ntrials:
  the number of trials to run the algorithm
- ndim:
  the dimensionality of the space in which the points are located (only used if flag is 2 or 3)

In minheap.cc, A min heap is a binary tree where each node is smaller than or equal to its children. The Heap class has several member functions:

- Heap::Heap(int mr):
  a constructor that takes an integer argument mr and initializes the maximum relevant member variable to mr.
- Heap::Heap():
  a constructor that initializes max_relevant to INT_MAX - 10000.
- void Heap::insert(tuple < int, double> v):
  inserts a tuple (int, double) into the heap. The integer is the identifier of the element and the double is the key value. The function also maintains the heap property by swapping elements as necessary.

- tuple < int, double> Heap::delmin():
  removes and returns the element with the smallest key value.
- tuple < int, double> Heap::peekmin():
  returns the element with the smallest key value without removing it from the heap.
- int Heap::size():
  returns the number of elements in the heap.
- void Heap::heapify(int i):
  reorders the elements in the heap starting from index i so that the heap property is maintained.
- int parent(int i):
  returns the index of the parent of the element at index i.
- int left(int i):
  returns the index of the left child of the element at index i.
- int right(int i):
  returns the index of the right child of the element at index i.
- int Heap::test_heap():
  a helper function to test the Heap class.

The implementation uses a vector to store the heap, with the root of the tree at index 0. The heapify function is a recursive implementation of the heapify operation. When an element is inserted, the heap property is maintained by swapping the element with its parent until the parent has a smaller key value or until the element becomes the root of the tree. If the number of elements in the heap exceeds max_relevant + 5000, the heap is resized to max_relevant.

The test_heap function tests the basic functionality of the Heap class by inserting three elements and checking that they are removed in the correct order.