CSCE 629 : Course Project

Sahan Suresh Alva, UIN - 130004855

November 24, 2019

1 Implementation

The project is implemented using Python 3. Packages like operator, random, math and time are used for the implementation.

1.1 Graph Generation

The graph is an object the user defined class called **Graph()**. The graph is implemented in a Adjacency list. Every list element represents a vertex and the points to a Linked List with vertices and weights it is connected to. Nodes of the Linked Lists are an object of **GraphNode()**.

The **GraphNode** object has the following attributes:

- vertex - Connecting vertex to the vertex of the list.
- weight - Weight of the edge between the two vertices
- next - Pointer to the next GraphNode in the linked list

The **Graph()** class has the following basic attributes and methods for Graph Generation:

- num_vertices : Initialize the graph with the required number of vertices
- addEdge(start_vertex, end_vertex, weight) : Add a weighted edge (GraphNode) between start_vertex and end_vertex in both the linked lists.
- removeAllEdges() : Removes all the edges of the graph
- createAllVertexLoop() : This method connects all the vertices of the graph in a loop. This is to ensure that the graph is connected.
- pair_generator() : This is a **generator** method. It generates an edge -a pair of two vertices in the graph. It makes sure that no two edges repeat and no self loop.

- createSparseGraph(avg_degree, max_weight) : Creates sparse graph of average degree 'avg_degree' by adding the required number of edges to the graph. First, the graph is connected by **createAllVertexLoop()**. Therefore, the additional edges to be added is calculated by this formula:

$$\#\text{ of additional edges} = \frac{(\text{average degree - 2}) * \#\text{ of vertices}}{2}$$

  The edges are generated randomly using the pair_generator method. The weights of the edges is chosen randomly between (0, max_weight).

- createDenseGraph(adjacency_fraction, max_weight): Creates dense graph of adjacency_fraction (i.e adjacency_fraction*100% of the vertices are connected to each vertex) by adding the required number of edges to the graph. First, the graph is connected by **createAllVertexLoop()**. Therefore, the additional edges to be added is calculated by this formula:

$$\#\text{ of additional edges} = \frac{\#\text{ of vertices} * (\#\text{ of vertices} - 1)}{2} * \text{adjacency fraction} - \#\text{ of vertices}$$

  The edges are generated randomly using the pair_generator method. The weights of the edges is chosen randomly between (0, max_weight).

1.2 Heap

The Heap is an object of the class Heap(). The Heap nodes are objects of the class HeapNode(). The Heap nodes are stored in a list to form a Heap structure.

The **HeapNode** object has the following attributes:

- name - Name value of the node (ex: vertex or edge)
- value - Value used to sort in the heap

The **Heap** object has the following attributes and methods:

- heap - List of all the HeapNode objects
- parentIndex(index) : Gives the parent index of the input index
- childrenIndices(index) : Gives the children's indices of the input index
- maxNode() : Returns the max of the heap
- insertHeapNode(name, value, $index_list$) : $CreatesaHeapNodewithnameandvalue, andinsertsittothe$ $DeletesaHeapNodeattheindex, andfixestheHeap.Anoptionalinputofindex_listisgiveninordertokeep$

- heapSort() : Performs HeapSort by doing maxNode() and deleteHeapNode() repeatedly. Returns the sorted by value node list.

1.3 Algorithms

All the algorithms are methods of the Graph() class:

- Dijkstra's without Heap - dijkstrasMaxBandwidthPath(source, destination, use_heap = False)
  - The dijkstra's algorithm is implemented without the heap. Here the max bandwidth value among the fringe vertices is found using a $O(n)$ method (linear search). Hence, the complexity is $O((m+n)n)$.
- Dijkstra's with Heap - dijkstrasMaxBandwidthPath(source, destination, use_heap = True)
  - The dijkstra's algorithm is implemented with the heap. Here the max bandwidth value among the fringe vertices is found using maxHeap() function. This takes $O(1)$. However, adding to n values to the heap and updating atmost m values in the heap results in the complexity of $O((m+n)logn)$.
- Kruskals's - kruskalsMaxBandwidthPath(source, destination)
  - The kruskals's algorithm is implemented with the the user defined heap. Here the edges are added and sorted using HeapSort. This takes $O(mlogn)$. This, along with UnionFind routine takes a total time of $O((m+n)logn)$.
- Kruskals's (using python sort) - kruskalsMaxBandwidthPathwithoutHeap(source, destination)
  - The kruskals's algorithm is implemented with the internal sorting function. The complexity remains the same. However, there is difference in the time elapsed due to internal implementations.

2 Results and Analysis

The program was tested on 5 pairs of graph with 5 different (s,t) pairs. The following table shows the average time taken in seconds to run the different algorithms in the experiment:

NOTE: All the elapsed time results are obtained from a local system (Macbook Pro). They are expected to vary proportionally on a different system with different configuration.

Algorithm	Sparse Graph	Dense Graph
Dijkstra's without heap	1.685	5.462
Dijkstra's with heap	0.177	2.874
Kruskal's	0.319	67.822
Kruskal's (using python sort)	0.423	9.804

2.1 Observations

- Sparse Graph:
  - You can clearly see that Dijkstra's with heap and Kruskal's performs very well as compared to Dijkstra's without heap. This is due to the fact that both run in the order of $O((m+n)logn)$
  - The obtained results are as expected.
- Dense Graph:
  - You can clearly see that Dijkstra's with heap performs better compared to all the other implementations. This is due to the fact it runs in the order of $O((m+n)logn)$
  - On contrary to our expectation, Kruskal's with heap takes significantly higher time than other algorithms even though it is of the order $O((m+n)logn)$. This is more of an implementation issue rather than the complexity.
  - You can clearly see the difference when Kruskal is implemented using default python sort as compared to user defined Heap.
- Combined Observations
  - Dijkstra's with heap is the best performing algorithm in both Dense and Sparse Graph.
  - Dijkstra's without heap shows expected behaviour in both graphs since it runs in the order of $O((m+n)n)$
  - Dijkstra's with heap performs better than the Kruskal's with heap due to the fact that there are fewer heap elements in Dijkstra's implementation (number of vertices) as compared to the number of heap elements in Kruskal's implementation (number of edges).

2.2 Kruskal's Implementation: Possible Issues

Kruskal's implementation should have been the fastest among all the algorithms since its complexity is lower compared to others. However, in this case it is significantly slower as compared to the rest of the algorithms especially in Dense Graph.

The the average time taken in seconds for different components of the algorithm graph as follows:

Component	Dense Graph (Heap)	Dense Graph (python sort)
Inserting all edges	15.78	2.9
Heap Sort	45.32	2.1
UnionFind	5.57	4.49
DFS	0.02	0.023

From the above results we can observe and infer the following points:

- The user defined Heap Operations are costly in **Python**.
- You can clearly see that Python sorted() - $O(nlogn)$ is much faster than the HeapSort() which is also an $O(nlogn)$ operation. This might be due to the internal implementations of **sorted()** in C (since it is an internal function) as compared to the the user defined HeapsSort function.
- However, this problem does not appear in sparse graph problems. This is mainly due to the reason that the Heap does not have large number of elements.
- This might be due to inefficient handling of the large lists in user defined functions in Python.
- Implementation of UnionFind functions are also slow due to this reason. (Large list in user defined function)

Clearly, due to the limitations of the programming language, we are not able to check the exact performance of the given algorithms. The internal implementation of the basic functions of Python are confounding the results obtained from the experiment.

3 Future Work

- We could implement the algorithms in a compiler based language like C or C++ to evaluate the exact results without confounding elements.
- If the goal is to find the Max Bandwidth of the target 't' only, then we can choose the stop the algorithms early when 't' gets added to the tree.
- We can implement a linear time algorithm for Max Bandwidth using the Median Finding and Connected Components algorithm.