Graph Clustering Using Alternative Distance Metrics

Abstract: This project explores graph clustering. It first presents novel distance metrics based on analyzing randomized spanning trees. A collection of these randomized spanning trees is generated using a technique similar to Kruskal’s algorithm. Then this collection forms the basis for measuring distances between nodes. One measure is based on the time it takes for two nodes to be connected during spanning tree creation. Another measure looks at the length of the paths that join nodes in the spanning trees. This second measure is then used in K means to perform the actual graph clustering. C, Python and Matlab were used for implementation. Test cases include dumbbell, planted, and image segmentation graphs. Results are inconclusive yet promising. Future work will involve clustering using the other distance metric as well as using clustering techniques other than K means.

Spanning Trees:

The entire project revolves around analyzing collections of random spanning trees (RST). Starting with a connected graph, G, with n nodes and m edges, the following method is used to generate a RST collection.\(^1\) The technique is very similar to a common algorithm used to generate minimum spanning trees (MSTs). MSTs minimize the sum of the weights of all the edges in the tree. Kruskal’s algorithm is a commonly used greedy

---

\(^1\) “Random” is an ambiguous term that may lead to some confusion. One understanding of random spanning tree would be to select each spanning tree from the collection of all spanning trees with a uniform probability distribution. Random walk algorithms can generate such spanning trees. However, this project uses a weaker meaning of random, and the technique presented here is more efficient because of this.
algorithm for generating MSTs. Kruskal’s algorithm works by building up a spanning tree from scratch. All the graphs edges are ordered from lowest to highest weight. Then each edge in turn is added to the spanning tree unless it would result in a cycle. This project’s RST generator is analogous to a version of Kruskal’s algorithm where the original order of edges is randomized.

The initial ordering is done using a Pseudo Random Number generator (PRN). It starts with an empty list and the list of all edges in an arbitrary ordering. It repeatedly uses the PRN to select and add edges to the randomized list until all edges have been selected. Once the edges have been ordered, it is just a matter of creating the spanning trees. The technique presented here uses the following three data structures: 1) Each node of the graph has a “parent” variable that is initialized as None.” 2) There is an array of components, in which each node begins in its own component list alone. 3) There is an array that maps each node to the appropriate cluster.

Every edge (connecting nodes v1, v2) then goes through the following process: look up the clusters, c1 and c2, corresponding to v1 and v2. If c1 = c2 then we know that v1 and v2 are already connected and that we must reject the edge. Otherwise, we determine the size of the two clusters. We then add the smaller cluster onto the larger cluster. I will assume that |c1| < |c2| without loss of generality. v1.parent is set to v2. For each node in c1, we reassign it to c2. Then the cluster list for c1 is tacked onto the cluster list for c2. C1 ceases to exist.

Once this process is over, each node will point to its parent in the desired RST. There are two main methods for storing RSTs. The first is to create an array of size n, where each element corresponds to a node. Each node then stores its parent in its spot in
the array.² A second less efficient but more general solution is to list out all the edges in the tree. By keeping all edges ordered according to when they were added to the RST, it is possible to back out the RST’s creation, which will be useful for one of the distance metrics.

This technique is complete for dealing with unweighted graphs. However, weighted graphs require a little extra consideration because it is necessary to select each edge with a different relative probability. To do this weighted probability selection efficiently we can keep all the edges in an array that lists each edge’s cumulative probability (the sum of all the edges’ probabilities through the current edge). Then once this is done, the PRN can select a value with uniform probability between 0.0 and the total of all the edge probabilities. Binary search can than select the appropriate edge in log time according to the desired probability distribution.

The one difficulty with edge selection is that the algorithm should never select the same edge twice. Recreating the cumulative probability array is a linear time proposition. So, it is impractical to update the array after each edge selection (which would result in a $O(m^2)$ algorithm). The alternative is to keep track of which edges were selected. Each time a PRN selects a previously chosen edge, the selection process must be repeated until an unselected one is chosen. However, as the algorithm progresses fewer and fewer edges are available and it takes more and more tries to find a viable edge. Therefore, this method is also too slow. A faster method combines these two approaches. Keep track of the fraction of unselected edges. When this fraction reaches a predetermined level then

² The root node is a special case that can be handled by having the root list itself as its parent. This requires that all software reading the tree recognize this convention and handle it appropriately.
recreate the cumulative probability edge array. If this limit is \(0.5\), then the array will only need to be redone a log number of times. Moreover, the expected number of tries to select an edge never goes above two, so the overall time to finish the ordering of the edges is \(O(m \log m)\) in expectation. Moreover, the rest of the algorithm follows Kruskal’s, which is \(O(n \log n)\). Therefore, the overall RST generation technique runs in \(O(m \log m)\) in expectation. The distance metrics in this paper require a whole collection of trees instead of just one. Such a collection requires that the above algorithm be run once for each of the \(t\) required trees.

Distance Metrics:

Once there is a collection of RSTs, it is possible to analyze them to measure distances between nodes in the original graph. The first method, Path Distance (PD) works in the following way: the distance between two nodes in the graph is the average distance between the two nodes in all the RSTs in the collection. This value is easily found by doing a breadth first search (BFS) from one node until the other one is reached. Because all the RSTs are by definition acyclic, it is unnecessary to use a priority queue as in Dijkstra’s shortest path algorithm. Thus, the distance between one node and all others is solvable in linear time.

The second method, Time Distance, measures the distance between two nodes in a graph in terms of how many edges have to be added to the RST before the two nodes are connected. This measure is easily normalized to always be in the range \([0.0, 1.0]\). For a collection of RSTs, the average over all the trees is taken. This value can be measured using the same techniques used to generate the RSTs. Given the ordered edges, rebuild up the spanning tree. The distance is simply the number of iterations until the two nodes are
in the same component. Thus, the distance between one node and all others is solvable in O(n log n) time. Future work will explore Time Distance, but the current focus of the project is on the applications of PD.

Clustering:

K means is a well-known technique for clustering. The idea is to create k clusters such as to minimize the intracluster distances. In this project, Lloyd’s algorithm is used to perform K means using the Path Distance metric (PD-Kmeans). Lloyd’s algorithm works iteratively. Given k clusters, find the center of each cluster, and rebuild all the clusters so that every datum is clustered its closest center. Repeat this process until it converges on a solution.

The code begins by randomly partitioning the nodes of the graph into k equal sized clusters. It then loads a RST collection to iterate through the reclustering process a user definable number of times. The tricky part is finding the center of each cluster. The original version of the code works by finding the single node in the graph that minimizes the sum of the Path Distance between it and all the other nodes. A better and faster method is to find the center of each cluster in each RST (each cluster might have different centers). The PD between a node and a cluster is the average PD to the corresponding center nodes in each RST.

Given a RST it is possible to determine the center node of a cluster in linear time. Each node has the following values: a boolean, C, that tells whether it is in the cluster or not, a list of children and its parent, weight propagating up and the number of nodes propagating it (WU & NU), and finally the weight propagating down and the number of
nodes propagating it down (WD & ND). All WU, NU, WD, and ND values are initialized to 0. I also assume that the weights of all edges are retrievable in constant time.

At the end of the process, \( u.WU + u.WD \) will be the total of the PDs from \( u \) to all other nodes in the cluster. We propagate the distances in two passes: first all the weights are brought up to the root; second all the weights are then brought back down. To do this we need to make sure that in the first pass no node is processed before its children and that in the second pass that no node is processed before its parent. This is easily assured by creating a list of nodes according to a BFS from the root. In the first pass, we process the all nodes in the reverse order of this list. In the second pass, we go through the list forwards. This list needs to be made only once for each RST and may be generated automatically if the trees data structure is built top down.

For each node \( u \) on the first pass:

\[
\text{u.Parent.NU} += \text{u.NU} + (\text{if u.C then 1 else 0}) \\
\text{u.Parent.WU} += \text{u.ParentNU} * \text{Weight(u, u.Parent)} + \text{u.WU}
\]

For each node \( u \) on the second pass:

For each \( v \) in \( u \).children:

\[
\text{vuNU} = \text{what was added to u.NU in the first pass from v} \\
\text{vuWU} = \text{what was added to u.WU in the first pass from u} \\
\text{v.ND} += \text{u.ND} + \text{u.NU} - \text{vuNU} + (\text{if u.C then 1 else 0}) \\
\text{v.WD} += \text{v.ND} * \text{Weight(u, v)} + \text{u.WD} + \text{u.WU} - \text{vuWU}
\]

Both passes are doable in linear time, and the node with the minimum propagated weight \( (WU + WD) \) can then be selected in linear time, so the overall algorithm to find the center is \( O(n) \).

Once all the cluster centers are found, it is possible to assign each node to the appropriate cluster. First create a counter initialized to 0 for each node-cluster pair. Then:

For each RST:

For each cluster, \( i \):

Run BFS starting at the center of the cluster.
As each node, \( u \), is found to be distance \( d \)

---

3 Note that the double for loop is linear in \( n \) because each node can be processed as a child at most once.
from the center do ctrs[u][i] += d
Now for each u in nodes:
    Find the min dist sum in ctr[u][:]. Add the node to the corresponding
cluster.
At this point, we have completed one iteration of PD-Kmeans. So if we do z iterations
with k clusters, n nodes, and t trees, it requires O(zknt) time.

Implementation Specifics:

Everything in this project runs on the linux zoo machines in the following three
languages: c, python, and Matlab. The code to create RST collections is in C and is the
fastest of all the components. The clustering all takes place in Python. Python was
originally a good choice to enable efficient experimentation with different distance
metrics and clustering techniques. However, Python is extremely slow and uses memory
very inefficiently. These performance issues limit the code’s use to relatively small
graphs (less than 50,000 nodes). Finally, Matlab code is used to get sample graphs as well
as to analyze some of the results. All clustering code is available on the project website.

Data is moved between different components using files in the following formats.
Graphs are encoded using an “ijv” format: The first two entries of the file specify the
number of nodes and edges in the graph. Then every edge is listed as (first node index,
second node index, 1.0/edge weight). These files are either encoded as ascii text with
each edge on its own line and every number separated by a space or as a binary file where
all node indices are encoded as C ints and all weights are C doubles. RST collections are
just concatenated ascii encoded ijav files (with edges ordered according to when they were
selected for their tree). A line is then added to the beginning of the file that specifies the
number of trees included in the file.

4 In order to conform to Matlab array indexing conventions node indices go from 1 to n
instead of 0 to n-1.
Clusters are encoded in a separate file format. First all clusters are arbitrarily indexed from 0 to k. The first line of the file specifies the number of nodes in the graph. Then each of the following lines contains the cluster index for the corresponding node. Using files to move data between different pieces of code is inefficient but makes putting together disparate pieces of code much easier. Moreover, in practice the time taken to write and read all the different files is very minor in comparison to all the other processes.

Results:

When clustering vertices in a graph, there are two main ways of judging success. The first is to look at the conductance of the resulting clusters, where conductance is measured as the total weight of all the edges between nodes in the cluster and those not in it divided by the total weight of all the edges adjacent to a node in the cluster. Assuming that clustering should divvy up the nodes in a graph into highly intra connected groups that are weakly interconnected, then better clusterings should result in smaller conductances.

Another approach is to evaluate a clustering against an ideal answer. Such is the case with the three main types of graph that have so far been tested. The first type is a 3d “dumbbell” graph, which is a graph of two 3d lattices of nodes connected together by a single path. Clearly, each side of the dumbbell graph should have its own cluster if doing 2 clusters. PD-Kmeans succeeds perfectly on such trivial cases (as does Kmeans using the usual measure of distance).

Another simple test case is a “planted” graph. Such a graph is created starting with two equally sized sets of vertices. Then two probabilities, p & q, are chosen such
that $p < q$. For each pair of vertices $(u, v)$ an edge is created with probability $p$ if $u$ and $v$ are in different starting sets or with probability $q$ if $u$ and $v$ are in the same starting sets. Because $p < q$, there should be more intra-set connections than inter-set connections. Thus, a good clustering technique should succeed in clustering all the nodes back into their original starting sets. Unfortunately, PD-Kmeans fails this test. A dataset is available on that shows the current results. More work is needed to understand this failure. Future experiments will use other clustering techniques to see if better results are achievable.

Images also present interesting clustering problems. If vertices represent pixels in an image and edges are used to describe the similarity in color between vertices, clustering can help divide an image into different parts as a human would. PD-Kmeans appears promising with such problems. PD-Kmeans successfully separated a person from the background sky in such a test. When finding more than two clusters, the results are more mixed. For example, the background sky is in one case broken up into two separate regions. However, most of the regions appear reasonable from the human perspective. The website shows the promising results of clustering a sample photo into two, three, and four clusters.

There are many performance issues with the current PD-Kmeans code mainly because of Python. Future work will require re-implementing much of the code in C++. It will also be useful to work on clustering using the other distance metric presented in this paper. Other researchers are also working on clustering algorithms that appear much more promising than Kmeans. It may be profitable to pair up these two distance metrics.
with other clustering algorithms, such as affinity propagation,\textsuperscript{5} to see if any improvements can be found. Finally, there has been limited time to experiment with different graph types. With more time, it will hopefully be possible to explore more types of problems, improve the current techniques, and better understand the strengths and weaknesses of the distance metrics. Finally, it will be interesting to explore uses other than graph clustering for the distance metrics.

\textsuperscript{5} See http://www.psi.toronto.edu/affinitypropagation/ for code as well as Science paper.