ISyE 6416 – Basic Statistical Methods – Spring 2016
Proposal

Team Member Names: Chris Meixell, Eisha Nathan, Chuanping Yu

Project Title: Choosing Between Branch-Based and Branch-Avoiding Algorithms for Social Network Graphs

Problem Statement

Graphs are a natural representation for modeling relationships between entities, whether in web traffic, financial transactions, computer networks, or society. For basic terminology, let G=(V,E) be an undirected graph, where V is the set of vertices and E the set of undirected edges with n=|V| and m=|E|, the number of vertices and edges respectively. Define an edge e=(u,v) connecting vertices u and v. Graphs can be used to model relationships between entities, where the vertices v in V can be thought of as players or users, and the edges e in E represent the relationships between these players. Assume the graph G is given in adjacency list format, i.e., for every vertex v, we have an array of its neighbors, denoted as "list(v)". Note that therefore degree(v) = size(list(v)). Since we are dealing with undirected graphs, each edge e=(u,v) in the graph will appear twice in the adjacency list representation, once in list(u) and once in list(v).

Specifically, we focus on those graphs modeling social networks. Much research of social networks focuses on identifying important players in a graph, using some centrality metric. One such metric is clustering coefficients, which is useful for finding key players in a network based on their local connectivity. This property is calculated using the number of triangles a vertex belongs to, making triangle counting an essential computation for social network analysis.

A very basic algorithm for triangle counting utilizes list intersections. Suppose we are given an edge e=(u,v). If vertices u and v have any common neighbors (denote the common neighbor as w), they will be found using a list intersection of list(u) and list(v). The triangle is then the one formed by vertices u, v, and w. We are presented with two algorithms for computing list intersections: a branch-based approach, and a branch-avoiding approach (as we will explain further below). The algorithm for triangle counting is then to iterate through every edge in the adjacency list, compute list intersection for the two vertices in question, and increase the triangle count for any vertex found in the intersection. Therefore, we will need to compute 2m list intersections, one for each edge as it appears in the adjacency list (note again the factor of 2 because each edge appears twice).

In many algorithms dealing with massive amounts of data, branch prediction is an important consideration when concerned with performance of these algorithms. "Branching" in an algorithm refers to the algorithm making a

choice to do one of two or more things; the simplest and most common programming example is the "if" statement. From a performance perspective, the presence of a conditional branch in an algorithm will interrupt the flow of instructions; if the algorithm does not know beforehand which branch to take, the processor will not know which instruction to fetch next, which will stall the pipeline, impeding fast runtimes. Therefore, previous research has explored branch-avoiding algorithms as alternates to a typically branch-based approach to many graph centric problems.

For our purposes, we are given two algorithms to calculate list intersections as mentioned above. For the scope of this project, we take the two algorithms as given and analyze the data resulting from execution times as explained below. Denote the branch-based approach as BB and the branch-avoiding approach as BA. Previous research shows that the branch-based approach is faster overall than the branch-avoiding approach for list intersections, where "total" refers to the total time for all 2m list intersections. However, examining individual execution times, where "individual" refers to the list intersection required for processing each edge, shows that approximately 30% of the time, the branch-avoiding approach outperforms the branch-based approach. This suggests there may be a hybrid approach of both BA and BB that achieve an optimal runtime for counting triangles. This gives rise to the question that we seek to answer: for an edge in the graph, can we predict which approach to take (BA or BB) to calculate the list intersection of the two corresponding vertices of the edge, given some parameters of the edge? For example, for an edge e=(u,v), possible parameters include size(list(u)) and size(list(v)). After compiling a list of these parameters for each edge, we wish to identify if we can predict which approach (BB or BA) will outperform the other and which approach would be ideal for a hybrid approach.

Data Source

We will use graphs from the DIMACS 10 Graph Challenge, which are widely used in social network analysis research. As a result of current research projects in the HPC lab in CSE, preexisting code will be used to generate the data. Since we assume the graph is in adjacency list format and each edge is appearing twice, we will have execution times for 2m edges, where m is again the number of edges in the graph.

As an example, the data may look like the following:

Edge	*Branch Avoiding (s)*	*Branch Based (s)*
e1=(u1,v1)	0.00123	0.00456
e2=(u2,v2)	0.00223	0.00111
...	...	...
Total	**0.0953**	**0.0843**

For each edge parsed in the adjacency list, we have execution times for both BA and BB for the respective list intersection as well as the total times for

each. In some cases we see that the individual execution time for BA is less than the corresponding one for BB, but the overall runtime of BB is faster than BA.

Additionally, as mentioned earlier in the Problem Statement section, we will have a set of parameters for each entry, i.e., features for each edge. Let "avg" denote the average degree in the graph. At present, we will consider features: (1) size(list(u)) -- integer, (2) size(list(v)) -- integer, (3) degree(u)<avg -- boolean, (4) degree(v)<avg -- boolean, but more features may be added later.

Methodology

-try a couple different classifiers

**linear classifier: tuning different parameters… try simulated annealing/iterative local search to pick best parameters?

-variable selection: which parameters actually matter?

-logistic regression

--not too technically oriented

--just overarching descriptions

Our methodology will mainly consist of implementing various classifiers and evaluating which model best fits our data. Before training the data to get the model, we can divide the data into two sets, training set, and test set. Here we may use different ways to get the training set and test set. Use the training set to obtain the regression models as stated below in descriptions of various models we may try, and then use the test set to see the accuracy of correctly finding the best algorithm for each model. The model that has the higher accuracy will be considered more effective than the others.

Linear regression

Let y1 be the execution time for the branch-based algorithm (BB) and y2 be the execution time for the branch-avoided algorithm (BA). Based on the features, x1, x2, …, xp, for each edge, we can get the regression model

$y1 = a0 + a1*x1 + a2*x2 + … + ap*xp$

$y2 = b0 + b1*x1 + b2*x2 + … + bp*xp$

Then, if y1-y2>0, BA is better than BB; if y1-y2=0, BA is the same as BB; if y1-y2<0, BB is better than BA.

Logistic regression

Let "z = 0" represent BA is better than BB, that is, the execution time of BA is less than that of BB; "z=1" represent BB is better than BA, that is, the execution time of BB is less than that of BA, and z follows Bernoulli(p). Then based on the features for each edge, we can get the logistic regression model

$p = h(c0 + c1*x1 + c2*x2 + … + cp*xp)$, where $h(x) = 1/(1+exp(-x))$.

If p>0.5, it means that the probability of "z=1" is greater than that of "z=0", so BB is better than BA; otherwise, BA is better than BB.

Support Vector Machine
Use the similar assumption as in the logistic regression: Let "z = 0" represent BA is better than BB, that is, the execution time of BA is less than that of BB; "z=1" represent BB is better than BA, that is, the execution time of BB is less than that of BA. Then based on the features for each edge, we can get the SVM classifier:

w1*x1+…+wp*xp – b >= 1 , then z = 1, that is, BB is better than BA;
w1*x1+…+wp*xp – b <= -1, then z = 0, that is, BA is better than BB.

Kernel SVM
Use the Kernel method to redo the above classification. We will try different Kernel functions and pick the best one among them.

Confidence Intervals
Once the parameters are fixed using the best model as evaluated above, we will attempt classification of an edge based on confidence intervals. Our tentative methodology will be as follows. Assuming an on-the-fly learning approach, we will parse the data edge by edge. Denote y1 as the execution time for the BB approach for an edge, and y2 as the execution time for the BA approach for an edge. Using these y_I's as `data,' we will calculate the confidence intervals at each stage (after each edge is processed). Essentially after edge i, we will have 2 confidence intervals: one for the estimation of the execution time for BA for edge i and one for the estimation for BB for edge i. We will continue estimating the intervals edge by edge until we have sufficient separation between the two confidence intervals. This will enable us to better predict algorithm times given an edge.

Additional Experiments
We will also consider any effects random sampling of edges and scanning order of the edges. For example, we will investigate if we obtain a better model, perhaps on less training data, by shuffling the order we parse the edges.

Expected Results
Our aim is to find a hybrid approach that, given an edge e=(u,v), chooses the fastest of the two list intersection algorithms BA and BB with as near to 100% accuracy as possible. Such accuracy could save significant execution time in finding the clustering coefficients of a graph's vertices. However, a hybrid approach with less than perfect accuracy could require more execution time than either BA or BB individually.
Define the variables below for some graph G with *m* edges. Also, let us assume that BA is faster than BB for 30% of the edges in G.

μ_{BA}=average execution time of BA on an edge in G.
μ_{BB}=average execution time of BB on an edge in G.
$\mu_{\Delta BA}$=average execution time savings when BA outperforms BB.
$\mu_{\Delta BB}$=average execution time savings when BB outperforms BA.

With 100% accuracy, a hybrid approach would be faster than BB overall by (0.6m)μΔBA since for 30% of the 2m executions, we would save μΔBA in execution time.
With 0% accuracy, a hybrid approach would be slower than BB overall by(1.4m)$\mu_{\Delta BB}$ since for 70% of the 2*m* executions, we would spend an additional μ*ΔBB* in execution time.
With accuracy between 0% and 100%, a hybrid approach would require between (2*m*)μ*BB* -(0.6m)μ*ΔBA* and (2*m*)μ_{BB}+(1.4m)$\mu\Delta_{BB}$in execution time, depending how accurately it chooses BA over BB versus BB over BA. For example, if $\mu_{\Delta BA}>\mu_{\Delta BB}$, then the best hybrid approach with X% accuracy would be the one that only inaccurately chooses BA over BB and never inaccurately chooses BB over BA. It is possible that, for most graphs, $\mu_{\Delta BA}$ and $\mu_{\Delta BB}$ will be statistically equivalent, but we have yet to explore this.
Additional future work will be application to large social networks. Although we will test our hybrid approach within graphs much smaller than those in many current social networks, we expect that our test graphs will be of sufficient size to not hinder future applicability. Increasing the size of a network graph should not affect edge-by-edge comparisons of BA and BB. It should only increase the number of edge-by-edge comparisons necessary to calculate the clustering coefficients of all vertices in the network graph.