Yisheng Zhou (Joseph)
Prof. Wayne B. Hayes
March 20th, 2016

Bio-Net Alignment Project Report

CS 199 Final Report, Winter 2016

Introduction

In the previous version, dual expanding Dijkstra's algorithm uses max heap as the data structure saving prospective pairs of aligned nodes. After applying random seeds into the algorithm, the program needs to pick a random pair whose similarity is in the range [max similarity minus given threshold, max similarity]. The previous version keeps popping out pairs with maximum similarity in the heap until the pop out pair has a similarity below a given value. Then, the program picks one pair randomly, and push every other pair back to the priority queue. Let k be the number of pairs whose similarities are in the given range. This progress takes T (k * log n + (k-1) * log n) time to push and pop, converting to O (k * log n). In Fall 2015 Quarter, I introduced a new data structure based on Skip List which can accelerate the progress of picking a random value in a given range from O (k log n) to O (log n).

However, this random method is not full randomness. The pipeline of the pop function is choosing a random value in the given range [max similarity – given range, max similarity], finding a pair with the nearest similarity of the random value and return this pair as the pop result. If the domain is [0.8,0.9], and the pair similarity set is {0.8, 0.849, 0.8495, 0.85, 0.851, 0.9}, the pairs whose similarities are {0.8495, 0.85} almost have no chance to be chosen, though they should own one third probability distribution to be chosen. This report introduces several methods including cumulative distribution function and reservoir sampling to make the pop function become fully randomness or faster, and lists several graphs showing the difference of the edge coverage results between full randomness and biased randomness.

Dataset

The program runs for three data files including "yeast.txt," "human.txt" and "sims.txt." "yeast.txt" saves the graph of yeast protein. "human.txt" saves the graph of human protein. These two files have a structure that every line is "Name_of_Node_1 Space Name_of_node_2," which means there exists an edge between node 1 and node 2. Name_of_Node is a string. "sims.txt" saves the similarity between a yeast node and a human node. It has a structure that every line "Name_of_Node_in_Yeast Space Name_of_Node_in_Human Space Similarity_Value." Similarity_Value is a float number between zero and one, while higher value means this pair of nodes have higher priority during graph expansion. In order to save time in checking whether a node is aligned or not and to save the space of storing aligned nodes, the program transfers the name of every node in two graphs to a distinct integer, uses the integer for calculation and makes two dictionary files. These two files are used to translate the integers back to strings. In the future, I will apply other kinds of data replacing "sims.txt," which changes the priority of graph expansion.

Technical Approach

1. Cumulative Distribution Function

One possible solution is to use the cumulative distribution function. This function reflects the probability that a random value is less than or equal to a given value, P(X<=x). The definition of function is F(x) = P(X<=x), and the curve of x and F(x) looks like a logistics regression curve. The pipeline of applying this function is picking a random value in the given range as a value in y-axis, projecting the value horizontally to the curve of the cumulative distribution function made from the similarity values of existing candidate nodes, and projecting down to the x-axis. The value on the x-axis is the value that the program will use to find the nearest similarity and return this pair as a pop result. After applying this pipeline, the program can almost achieve full randomness.

However, the progress of making the cumulative distribution function costs a huge amount of time. NUMPY has a build-in CDF. It needs to take all values for calculation, which means it may take O (n) time, while our data structure needs to have a O (log n) time. The CDF function is just a curve. When we insert or delete a value, it might be updated by just multiplying or dividing something. I try to find an O (1) or O (log n) method to update the CDF function after inserting or deleting a pair from candidate node set but does not have a solution right now.

2. Counter

Rather than applying a real accurate CDF, I try to make an erroneous but practical CDF function. If the error is small enough, I can still get reliable random values because the calculation of similarity value is also erroneous. My idea is to use a Global Counter, which is like a histogram recording how many values in a given range (for instance, 0.9998_0.9999 is one range). In this way, every time the program inserts or deletes a variable, it only needs to change one value in the counter. When the program needs the CDF, use the values in Counter to get a virtual CDF. This is not a real distribution based randomness, but it could be a practical method. The time can be reduced to a limited constant number based on the size of each range. The resolution of the counter can be kept in some acceptable rate because the Priority Queue does not save everything in similarity file. It only saves the pairs of node that our current extended graph can access.

3. Reservoir Sampling

While applying CDF function, I also keep thinking other slow but simple methods to achieve full randomness. The previous method takes O (k * log n) to get a random value because it needs to form a new list and pick a random value from the list by choosing a random index. The skip list already sorts all value as a linked list so I can apply Reservoir Sampling to pick a random value from the first several elements in a given range. The program goes through the linked list and determine whether takes value of the current node as the temporary result. When the next value is out of our required range, the progress stops and returns the temporary result as our final result. This progress will take O (k) time to get the random value. It reduces the process to an acceptable time length but is still slower than O (log n) pop.

Software

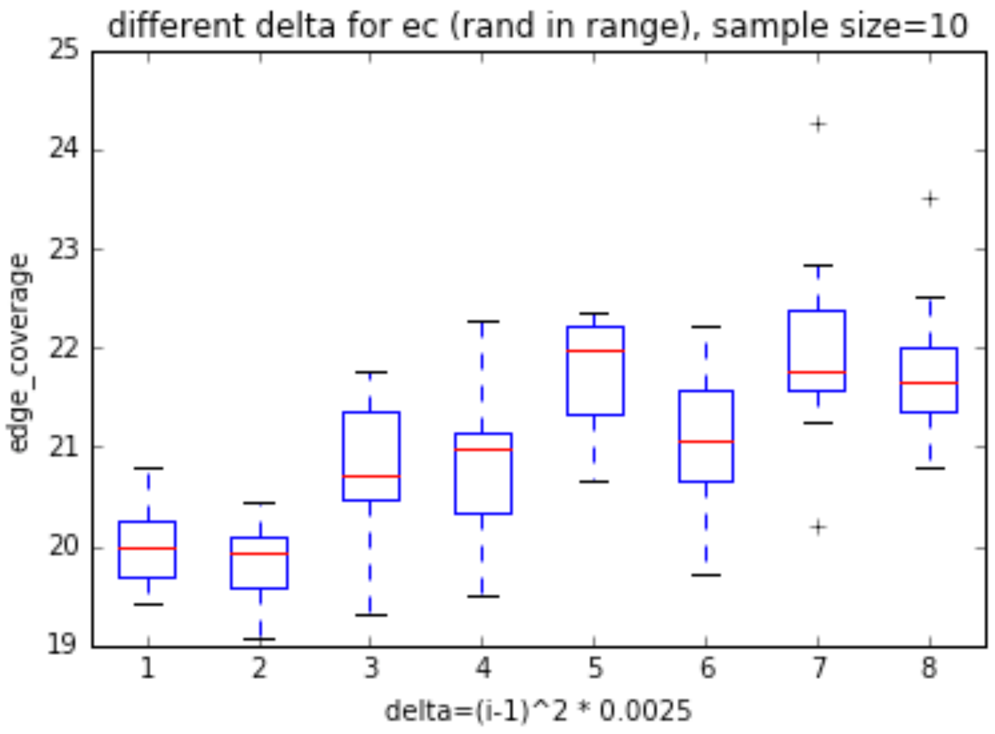
The following calculation is based on Python 3.5 code. The IDE is Spyder version 2.38 from Anaconda Package 1.0.0. I use matplot function in Python matplotlib to draw the following graph. The random functions are from Python Random library.

For all of the following graphs, I compressed all of the results into a zip file. Every file in this zip saves the EC and the alignments. The file name includes the delta, the length of running time in sec, and the finish time, which is the primary key. The alignments are shown as integers because it requires extra running time to translate the integer into string (protein name).

Evaluation

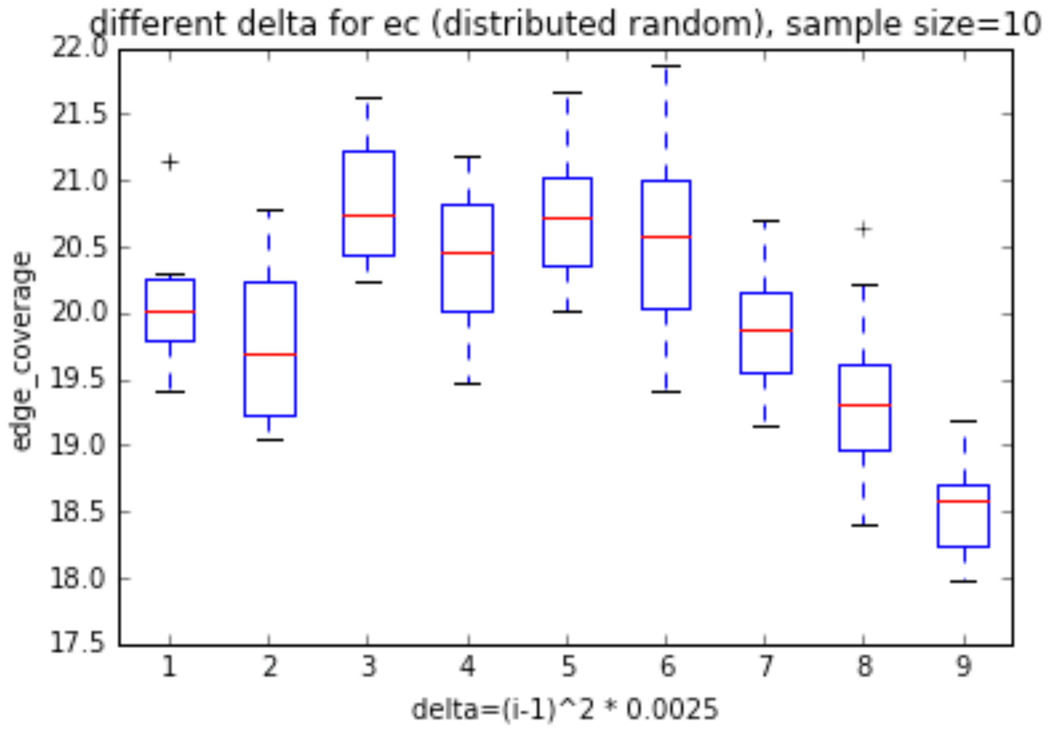
This part is used to determine whether the full randomness matters. It includes the boxplot graphs showing the relationship between delta and performance and the line chart showing the relationship between delta and running time. It includes the O (log n) method mentioned in the introduction and O (k) method applying Reservoir Sampling. I choose boxplot because it is a convenient way of graphically depicting groups of numerical data through their quartiles. It can reflect the average value, the standard deviation and some extreme results at the same time. The evaluation function is edge coverage. My assumption is that although we give the inside variables more chance to be picked, the edge coverage could still keep the same. But, I believe the standard deviation can be larger because the program expands in more different ways.

1. Reservoir Sampling

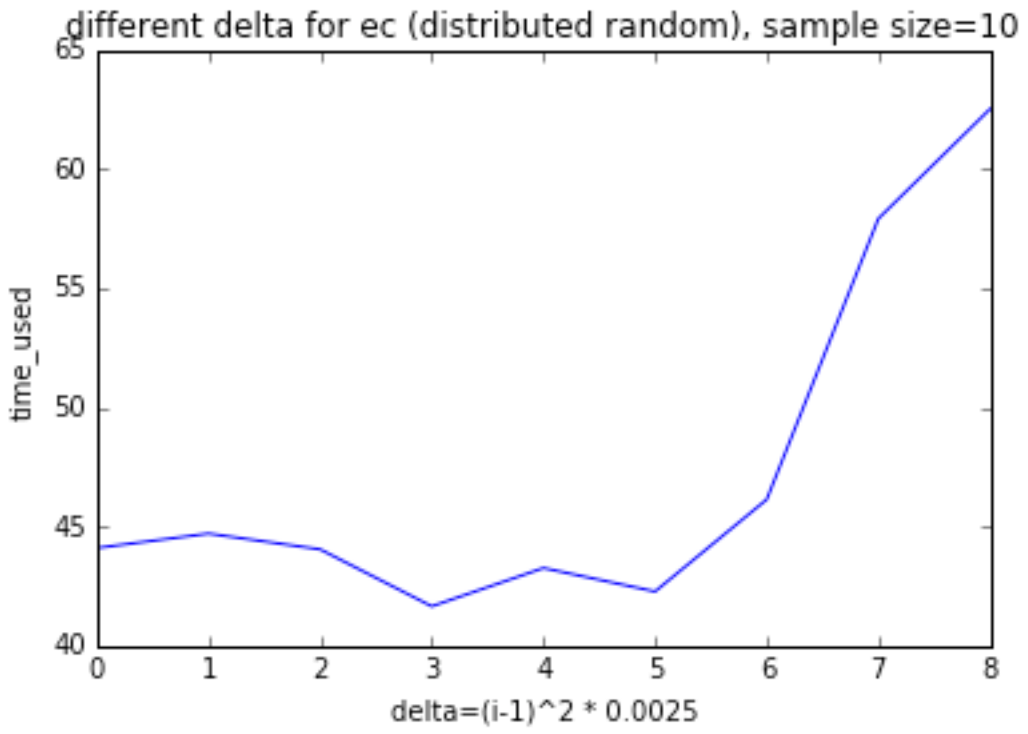


The average time usage for each delta is [40.4625087184303, 40.441319907271392, 39.445512494029138, 65.538055974995245, 289.42997882773835, 1595.9111788344094, 5102.3771437930545, 10636.662779454822]. They are measured in seconds. The range of the time usages is sparse, so I did not plot the list in graph. Also, when the delta reaches 0.16, it almost takes half of a day to run. For the first three, I assume there are several pairs having same similarities values, and the cost of random time does not determine the entire running time. There exist some extreme results. In this 80 runs, the best EC is 24.26%, when delta = 0.09.

2. Not fully randomness



The following graph showing the time usage of different delta. Because of some typo, please regard the X-ray label as "delta=i^2 *0.0025."



Conclusion

Based on the graphs in the evaluation section, we can make two conclusions. Firstly, the full randomness does improve the performance of our Dijkstra's Algorithm. Secondly, the Reservoir Sampling pop costs more time than the O (log n) pop. The current results do not reach the turning point of the full randomness pop because when delta reaches 0.16, the program with O (k) pop almost takes half of a day to get one result. Therefore, if possible, I should apply methods such as CDF function into O (log n) method to get a faster pop with full randomness. In my C++ code, I keep two kinds of pop. If the program wants to get higher edge coverage, it should call pop1, which uses Reservoir Sampling to get the random value. If the program needs to get a large set of results in a restricted length of time, it should call pop2, which applies random-in-range method.