Richard Pham
Final Project Write-Up
Professor : Anna Ritz
5/10/2017

Overlapping to Order

The main point of this project was to obtain the correct eulerian path from an overlap graph of kmer-edges. I further split this question into four sub-problems : parallel edges can exist and cannot exist, and duplicate edges can exist and cannot exist. Parallel edges are defined as such : if edge := (u, v), then parallel edge := (v, u).

I was inspired to embark on this venture because of the graph theory knowledge I gained during this semester, in this class as well as others. During this time of inspiration, I took many tangents, whether to debug code, to brainstorm (which took a sizable portion of my time on this project), or study/research. As a result, I do not believe my project is finished in any way, shape, or form, the fact which I will elaborate upon below.

I first implemented a graph class, alongside with minimum spanning tree, graph node, and graph edge. My graph class includes constructor methods. These methods are used to add to the class attributes “nodes” and “edges”, as well as to construct an adjacency list, which consists of elements such that each element is a list, where the first element is some node value and the second element is a list of adjacent node values. There are also searcher methods, which given some value, searches class attributes “nodes”, “edges”, and “adjacency_list”, fundamental for the program to determine what to do given some edge or node, such as in the problem of finding a next “safe” edge. My graph class also has the two standard traversal methods : breadth-first search (“bfs_op”) and depth-first search (“dfs_”). In order for these two traversal functions to be implemented, I had to add to my “GraphNode” class the attributes “color”, “distance”, “start”(start-time), and “finish”(end-time). There are also default methods, which default the attribute values of my Graph class’ attributes “nodes” and “edges”. Finally, there is a sort method called “order_edges_ascending” which uses my implementation of the universally-known mergesort to order edges according to non-decreasing weight. This method is not directly used for functions in my Graph class, but is essential for algorithms such as Kruskal’s and Dijkstra’s .

Speaking of Kruskal’s algorithm and Dijkstra’s algorithm, I also implemented these two algorithms, each of which are separate files in my project source folder. This decision was made during my time of brainstorming on how to guarantee a correct Eulerian path given no parallel edges. I also implemented the Boyer-Moore string search algorithm. Although this string-search might not be used directly for finding Eulerian paths, I am sure it has application somewhere in this problem.

Now, with most of the prerequisite code covered, I will cover the two Eulerian path/cycle-finding algorithms I implemented, Fleury’s Algorithm and Hierholzer’s Algorithm. Fleury’s algorithm, although intended for undirected graphs, seems to nevertheless be relatively accurate. I implemented these algorithms in various versions, as indicated by the number of Fleury and Hierholzer files in my source and scratch folders, in the hopes of finding a good modification to raising the accuracy of my Eulerian path. In my original versions, I used a default choice given possible next valid edges, always picking the first valid edge. In my second approach, I used a randomized scheme to achieve the same objective. For instance, given a choice of valid edges A, B, and C, my second approach would have a equal chance for choosing any of the three options. I was able to make my Fleury’s algorithm less volatile, that is, it returns a reassembled sequence of correct length a great bulk of the time, along with results that are not 0% accurate. This

satisfactory performance is directly opposed to that of Hierholzer's algorithm, which performed extremely well in some cases, but 0% accurate in others, despite the next-edge-option scheme (randomized or default).

On the issue of ALL_EULERIAN_PATHS(), I was not able to complete a working implementation of it; my approach was resource-expensive, and I also took a turn in this project to study cluster computing designs, which of course, did not directly benefit my project. My theoretical approach towards ALL_EULERIAN_PATHS() is as follows : for every possible start kmer node, I would calculate an initial path, recording indices in this path where 2 or more options exist. For instance, given the path ['ACTGA', 'CTGAA', 'TGAAC', 'GAACA', 'AACAT', 'ACATT', 'CATTA', 'ATTAA'] the options index list could be the following : [0, 1, 4], where 'ACTGA', 'CTGAA', and 'AACAT' are the elements where an alternative next node exists. Then I would backtrack through indices 4, 1, and 0, at each index recreating an alternative path. The primary issue here is that in any significantly-sized sequence, there would be a multitude of options list, one for the initial path, and others for each recreation of the path at some options index. I tried to find an efficient specification for this theoretical approach, but I continually stumbled upon the problem of options index list.

Testing

On to the testing : I have two main files, one called "_main_.py" (the primary main file), and a previous one called "main_.py", which I catered towards the subproblem of no parallel or duplicate edges. This condition of no parallel or duplicate edges resulted in highly inaccurate reassembled sequences and even though I attempted to incorporate an edge-repeat attribute to compensate, this was to no avail; I embarked on this problem because I found it quite challenging, challenging to the point where it is virtually impossible. It was a fool's errand, so I dismissed working on the "main_.py" file and focused on the other simpler subproblem : that duplicate and parallel edges can exist in the graph. The results can be ascertained through the "_main_.py" file, simply run : autorecord_all() for the program to create a set of files where results of accuracy and reassembled sequences are found.

The following is an overview on the operations of autorecord_all(). Given some string that is converted into an overlap graph, autorecord_all() churns out 4 files for every string. One file is an accuracy-test for both Fleury's and Hierholzer's algorithms (comparing the reassembled sequences of these algorithms to the actual seqeunce), one is an accuracy-test for only Fleury's algorithm, another is an accuracy-test for only Hierholzer's algorithm, and a test-collect file which returns all the reassembled sequences of Fleury's and Hierholzer's algorithms.

Results

I used four simple test strings (found in variable strings_list in "test_strings.py"):
'AACTAACTATATG', 'ACTGGGACTTTAA', 'AACTCGCGCAGAGA', 'TTAGCACATAGATAGATA'.

I will give a personal summary of the accuracy results I obtained through my last run of autorecord_all():

- string 'AACTAACTATATG' :
  - accuracy test (both Fleury's and Hierholzer's) :
    - 30.82 %
  - accuracy test (only Fleury's) :
    - 11.15 %
  - accuracy test (only Hierholzer's):
    - 50.97 %

- string 'ACTGGGACTTTAA' :
  - accuracy test (both Fleury's and Hierholzer's) :
    - 75.22 %
  - accuracy test (only Fleury's) :
    - 49.78 %
  - accuracy test (only Hierholzer's):
    - 100.0 %
- string 'AACTCGCGCAGAGA' :
  - accuracy test (both Fleury's and Hierholzer's) :
    - 75.125 %
  - accuracy test (only Fleury's) :
    - 49.81 %
  - accuracy test (only Hierholzer's):
    - 100.0 %
- string 'TTAGCACATAGATAGATA' :
  - accuracy test (both Fleury's and Hierholzer's) :
    - 12.84 %
  - accuracy test (only Fleury's) :
    - 25.77 %
  - accuracy test (only Hierholzer's):
    - 0 %

These results may have high accuracy percentages, but they highlight a persistent problem throughout my work on this project : the volatile nature of Hierholzer's algorithm. Observe the accuracy results for the last string. Hierholzer's algorithm scored a 0 % on it. I mentioned that I implemented several versions of Fleury's and Hierholzer's algorithms, but all my randomized next-choice-for-path approaches to Hierholzer's algorithm (files are "hier_2.py" and "hier_3.py") churn out completely wrong results. And so, I stuck with the default next-choice-for-path option, that is, the file "hier_.py".

If you were to personally run autorecord_all(), I doubt you would have the high percentage results I encountered.

Findings

Fleury's algorithm, although intended for undirected graphs, seems to do a better job of consistently returning paths of correct length, as per my implementations. Hierholzer's algorithm, however, does not. Recall that the condition for finding a Eulerian path via Fleury's algorithm is that there are either 0 or 2 odd-degree nodes, and that for Hierholzer's algorithm is that every node has an equal in-degree and out-degree. The condition for Hierholzer's algorithm is what seems to return false for some of my test strings. Whereas Fleury's algorithm is intended to retrieve paths, Hierholzer's is to ascertain some Eulerian cycle. This stricter condition seems to impede on its accuracy. As a result, my "hier_.py" file is a slight modification on the original Hierholzer's : it abides by the same condition as Fleury's, instead of the equal in-degree/out-degree condition. This may seem flawed, but this modified Hierholzer's algorithm, "hier_.py", seems to be more accurate than that of "hier_2.py" and "hier_3.py".

A persistent problem towards more accuracy is the next-edge-option scheme. This scheme has two variants. Given a list of next-edge options, one variant chooses the default first one that qualifies for some condtion, the other chooses a random edge from this list that qualifies.

Thoughts for Future Work

If I were to do this project over again, one thing I would change is the proportion

of brainstorming/researching I conducted on how to apply graph traversal algorithms such as Dijkstra's to finding an accurate Eulerian path. These activities took the great bulk of my time. My fascination with computing seems to have done me little good in bringing together a refined product, rather it split my time and diverted my focus.

Instead, I would have taken a closer look into Fleury's and especially Hierholzer's algorithms. My implementations were not up to my expectations. I coded several versions of these two algorithms, but I seem to be replicating the same buggy code each time. I cannot state any specific issue, but I would attribute these buggy implementations to my misdirected energies in coding, in other words, I need to do more testing at smaller intervals during the coding process.

After I have better refined my Eulerian path-finder algorithms, I will probably take a more meticulous look at graph theory concepts as well as genomic sequencing machines, so that my code will not be so duplicate and lacking in fundamental understanding. The problem of ALL_EULERIAN_PATHS() will, for the time-being, be a work in progress.

Other interesting ideas that I stumbled upon in this venture include assigning edges to some version of an overlap graph to increase the accuracy of retrieving the correct assembly. For instance, given some background info along with the kmer edges, then we could assign weights to the edges such as path will take the accurate next edge given more than one options. I have not devised a working logistic for this plan.