Grady Yu
93162142

CS 199 Final Report Fall 2015
Professor Wayne Hayes

Introduction:

Our research group's objective was to discover a good method for matching common parts of the protein network of yeast and the protein network of humans. The chosen algorithmic approach was inspired by Dijkstra's minimum spanning tree algorithm. This quarter, we modified our definition of the set of neighbors and adjusted our internal data structures to accommodate.

Summary of Results:

This version of the algorithm produces an alignment with 20% yeast coverage and 7% human coverage. This result marks a remarkable improvement since Spring quarter, which only had ~11% and ~4% respective coverages.

Contributors: Grady Yu, Yisheng (Joseph) Zhou

Description of Deliverables:

Only the changes from last quarter are included in this report. Please see the previous quarter's report for a full description.

Neighbors defined: In Dijkstra's algorithm, the set of neighbors includes any node adjacent to any already visited node. However, our problem has two graphs and cannot follow this exact definition. A natural analogue would be the the cross product of the neighbors of each graph.[1] This definition captures every pair of nodes where each node is adjacent to the alignment of each graph.

However, many of the pairs in $N_Y \times N_H$ are poor choices for building a maximally aligned subgraph because they pair nodes which are adjacent to topologically distant nodes in the alignment. Since the algorithm picks the best pair by highest similarity, a highly similar pair of nodes which are topologically distant in the current alignment would have precedence over a topologically favorable but less similar pair. This type of pairing can contribute to a poor alignment since the nodes aligned are too far apart in their respective protein networks to be analogues. Therefore, the neighbors should be constructed via a more narrow definition of adjacency that filters out these bad pairs.

The desired set of neighbors resembles the natural join rather than the cross product. $N_Y \bowtie N_H$ includes only the pairs of $N_Y \times N_H$ where each pair is adjacent to a pair in the alignment. This removes the pairs with disjoint neighbors from the set and prevents topologically unrelated nodes from being aligned. This definition increases the influence of local topology on the decision of the next best pair.

Defining the set of neighboring pairs as the union of the cross product of the neighbors of each individual aligned pair[2] allows the algorithm to incrementally construct the set. Since the pairs are added to the alignment one at a time, the set of neighboring pairs will be the union of the previous neighbors and the new pairs neighboring the current node. As this process constructs all intermediate sets of neighbors, it runs quite efficiently.

Near matching: Since the similarity measures provided are derived from inexact laboratory measurements, not every figure in the measurement is significant. In addition to all equally similar

1 To be precise, $N_Y \times N_H = \{\{n_Y, n_H\} \forall\, n_Y \in N(A(Y)), \forall\, n_H \in N(A(H))\}$, N() is neighbors, A() is alignment, n is node in y

2** $N_Y \bowtie N_H = \{\cup\, N(y) \times N(h) \forall\, N(y) \mid y \in N(A(Y)), \forall\, N(h) \mid h \in N(A(H))\}$

pairs, considering pairs within some epsilon measure of the top similarity score expands the search space greatly. The increase in possible states allows the exploration of more potential alignments.

To avoid partitioning the pairs, this must be done dynamically. The best pair algorithm should accept any pair with similarity greater than or equal to (top – ε). The “or equal to” must be included to properly handle ε = 0 and it prevents other cases which cause the algorithm to fail. Static methods, like rounding, would cut the pairs along undesired boundaries (e.g. 0.954 is within ε = 0.01 of 0.095 but 0.944 is not within ε = 0.01 0.952 after rounding to 0.01).

Data Structures: The best pair must be selected from the global set of neighboring pairs. Unlike the previous version, where the best pair could be quickly recomputed after each step, this version computes the best pair state-fully. The neighbors must be stored in a data structure that allows quick access to the most similar pair. This requirement hints strongly that an abstract priority queue would be an efficient choice.

However, a good implementation of the abstract idea has been difficult to write. As a lazy placeholder, a list sorted after each insertion took ~10 minutes on ε >= 0.01 and seemed to increase exponentially with higher ε. A binary heap was difficult to write and test successfully because the default libraries would not accept data of the desired format. After finishing the heap, the runtime was about half of the sorted list, hinting that the runtime may be dominated by the Ω(n log n) lower bound for sorting.

Since each additional node adds many potential best pairs, the data structure must support efficient multiple inserts and quick access to the top n elements for variable n. A sorted list met these requirements, as a pre-pend could insert all elements and cutting the list at element n would return the top n elements. However, the cost of sorting after insert eclipsed the savings from these steps. The heap had neither of these properties. Eventually, our group settled on a skip list because it could most efficiently implement these properties.

Because of oddness with comparison operations on numpy arrays, the data must be stored within a wrapper class. This may cost some performance but not enough to negate the gains of incorporating the library if comparisons are minimized. Additionally, any data structures should minimize movement of elements, as this takes a relatively long time in Python.

Change in best pair near randomization: According to Yisheng, selecting from the n top elements will be awkward in the skip list. So, he has replaced select randomly from n with an alternate method based on the similarity values. The algorithm picks a random value in the acceptable range of ε and searches for the last pair to meet this value. This almost certainly skews the probabilities such that all pairs are no longer equally likely. As long as the probability distribution is well behaved, this should not be a problem.

Until we have a target distribution, we only need to avoid distributions with bad properties. No pair should have virtually no chance of selection, relative to the number of pairs. No pair should dominate with an exceptionally high chance of selection. As long as these two conditions are met, this method of randomized selection may perform acceptably.

changes to files: The data structure will be in a separate file.

Conclusion:

This version of the code has greatly improved over the previous due to the change in the processing of

neighbors. It has been challenging to efficiently implement this process but the new code has greater potential. With this done, the next step would be to update the similarity measure to incorporate new data that has improved the measures.