Grady Yu
93162142

CS 199 Final Report Spring 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 introduced some randomness into the algorithm to improve the quality of the results.

Summary of Results:

The best coverage the current version of the algorithm produces aligns ~11% of total yeast edges and ~4% of total human edges. This result marks a remarkable improvement over last quarter's run, which only had ~3% and ~1% respective coverages.

This best result was observed from a small handful of runs. In theory, there should be a very large number of possible alignments that the algorithm can output so the algorithm has the potential to find a better result.

Contributors: Grady Yu, Yisheng (Joseph) Zhou

Dependencies:

Numpy: The current version of this program depends upon the Numpy library. This enables a huge speedup as well as a reduction in memory usage.

Changes:

- build_sim() creates a Numpy array instead of a Python list of Python lists. This supposedly takes less space. The runtime of this function did not change.
- best_pair() has been rewritten for faster access to a Numpy array.

Note: The Numpy array can be indexed like the previous sims array but at the cost of a significant performance penalty.

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.

sims: After some experimentation, reversing the “similarity” value in the sims file seems to lead to a better result. We have strong evidence that this “similarity” file is actually a “difference” file and apply a transform (1-x) to get the similarity value.

Multiple files: This program has been split across three files for better modularity.

- *graph.py*: holds the Graph class and its associated functions.
- *builder.py*: holds functions for constructing data structures like the similarity matrix and the seeds generator.
- *research.py*: holds the functions for algorithmic computation like best_pair() and dijkstra().

Randomization: The sims file contains many pairs of equal similarity. Previously, this tie was broken by choosing the first one in the file. Now, this program picks randomly from all the nodes with equal similarity.

get_seed() takes a file as input and returns a generator that yields pairs of yeast and human nodes. The input file must be sorted in **descending** order of similarity.
pseudo-code:

```
tied_seeds = []
curr_value = 1.0
while sims is not empty:
    row = next seed
    if row.similarity < curr_value:
        yield every seed in tied_seeds in randomized order
        empty tied_seeds
    add row to tied_seeds
//handle tail case
yield every seed in tied_seeds in randomized order
```

This algorithm's randomness comes from shuffling the array of equally similar seeds. Since each seed should only be yielded once, a random selection from the array would not be correct, as a truly random selection should be able to select the same item twice.

best_pair() returns the best pair of nodes within a subset of yeast and a subset of human nodes. If there are multiple candidates, best_pair selects one at random.
Runtime: O(|y| * |h|) where |y| and |h| are the number of rows and columns specified in the arguments. As this function must run for every step of dijkstra(), its total runtime is O(|y| * |h| * |y|) ~ O(n^3). Practically, the default Python version took ~90% of the total runtime! The Numpy version has reduced this to ~50%. This represents a 10-fold speedup.

Conceptual pseudo-code:

```
yeast ← the subset of yeast nodes
human ← the subset of human nodes
sims ← a yeast X human matrix of similarities
output ← the best pair from the subset of yeast X human
best ← the set of best pairs, given the current state of the algorithm
for y in yeast:
    for h in human:
        if sims[y][h] > best.value():
            best = (y, h, sims[y][h])
        else:
            best.add((y, h, sims[y][h]))
return a random choice from best
```

Implementation pseudo-code:

```
//variables are the same as before
convert yeast, human from sets to lists //because of Numpy
create a sub-matrix of sims containing only the rows of yeast and columns of humans
filter this sub-matrix for all the pairs of equal and highest value.
return a random choice from that set
```

Since each call to this function returns only one pair, a random choice will suffice.

induced_subgraph() has been rewritten to resolve a bug where some pairs could be double counted.
Pseudo-code:

```
aligned_pairs ← the set of pairs aligned by the Dijkstra's algorithm
graph1, graph2 ← the corresponding graphs of the aligned pairs
result = []
while aligned_pairs is not empty:
    p = aligned_pairs.pop() //treat as a queue
    for q in aligned_pairs: //this does not include p
        if graph1.has_edge(p,q) and graph2.has_edge(p,q):
            add this to the result
return result
```

Future directions:
Multi-seeding: The current algorithm adds seeds to the graph one-at-a-time. Adding more than one would increase the number of possible pairs and possibly allow for better alignments.

Multi-threading: Now that the algorithm makes random selections, each run will produce a different result. The program should be modified to generate more than one alignment at a time in order to explore more of the possible alignments.

Conclusion:
The randomness should be correctly implemented into this version of the code, given that the Python random module works as expected. This has increased the possible alignments that this algorithm can discover, allowing it to be run repeatedly in search of a better result. A few ideas did not get implemented this quarter but are good next steps.