SANA Fall Report 2019-2020

CS 199 - SANA group
Utsav Jain, Tingyin Ding

Abstract:

Previously, the Dijkstra aligner used an internal mechanism for seeding the initial alignment. This quarter, we started to use an algorithm which produces database "keys" that can serve a seeds to a local alignment. Besides, in order to produce perfect EC score and S3 score alignments, the aligner must follow a strict criteria when aligning each pair. The Dijkstra aligner now accepts the lowest boundary for EC1 and EC2 and a weight between similarity score and edge coverage as criteria.

Introduction:

The Dijkstra aligner used the similarity file to get the highest similarity value pair as the next seeding pair. In the beginning of the quarter, we added some randomness to the Dijkstra seeding selection based on the similarity file. Then, instead of using the similarity, we used the basic graphlets as the starting aligned subgraph. Initially, we modified the algorithm to use degree difference as the restriction for expanding pairs. However, degree difference did not seem to produce good alignments, so we then modified algorithm to use edge coverage as criteria to align. Finally, we added command line arguments so the user can specify the minimum EC1, EC2, or S3 score needed for alignment. If during the alignment, there are no pairs that satisfy the lower bound set by the user, the program stops aligning. The command line can also accept a weight between similarity score and edge coverage.

Methods:

With the initial seeds from the basic graphlets found by BLANT, the first version aligned strictly that only expanded the pair with the same degree. The second draft popped out the pair with the highest similarity score and maintained the same degree of two nodes in the aligned subgraph. Finally, instead of just maximizing ec1, we set the lowest boundary for ec1, ec2, or s3 and combined it with similarity scores with given weight alpha.

Current Implementation:

After the initial seed graphlet is aligned. The main while loop starts. Currently, the candidate pairs are added to a list (python array). The list is then sorted by each pair's M value, meaning we are trying to align pairs which increase EA the most first. We then iterate from the end of the candidate pairs list, popping off each pair. If the pair satisfies this requirement ((EA + mval)/(E1 + n1val)) >= ec1 and ((EA + mval)/(E2 + n2val)) >= ec2, where ec1 and ec2 are the lower EC1 and EC2 bounds set by the user, then the pair is aligned. If not, we reserve this pair since it may satisfy the requirement, so we add it back to candidate pairs in the end of the current loop.

The pseudo code is below:

N1 is the number of edges from g1node back to graph1 in current aligned subgraph S.
N2 is the number of edges from g2node back to graph2 in current aligned subgraph S
M is number of edge pairs from pair (g1node, g2node) back to current aligned subgraph S.

E1 is number of aligned edges from Graph 1.
E2 is number of aligned edges from Graph 2.
EA is number of aligned edge pairs.

```
local_align(graph1, graph2, seed, sims, ec_mode, m, delta, alpha):
    #m is number of edges in seed graphlet
    E1 = E2 = EA = m
    g1alignednodes = set()
    g2alignednodes = set()
    aligned_pairs = set()
    #candidatePairs = all pairs (u,v) within 1 step of S, u in G1, v in G2.
    initialize candidatePairs as an empty array
    initialize edge_freq to save the n1, n2, M for a node pair

    #align initial seed graphlet
    for seed1, seed2 in seed:
        align (seed1, seed2)
        add all combinations of seed1's neighbors and seed2's neighbors to candidatePairs

    done = False
    while(not done) do
        done = True
        new_candidatePairs = []

        for g1node, g2node in candidatePairs:
            if (g1node, g2node) not in edge_freq and g1node not aligned and g2node not aligned then
            update N1, N2, M for (g1node, g2node) to edge_freq

        sort candidatePairs smallest to largest by following
        if ec1 > 0 and ec2 == 0 then
            sort by alpha*(M/N1 - ec1) + (1-alpha)*similarity_score
        else if ec2 > 0 and ec1 == 0 then
            sort by alpha*(M/N2 - ec2) + (1-alpha)*similarity_score
        else if ec1 > 0 and ec2 > 0 then
            sort by alpha*(M/(N1+N2)) + (1-alpha)*similarity_score

        while candidatePairs is not empty do
            (node1, node2) = candidatePairs.pop()
            if node1 not aligned and node2 not aligned then
                if ((EA + M)/(E1 + N1)) >= ec1 and ((EA + M)/(E2 + N2)) >= ec2 then
                    align (node1, node2)
                    add all combinations of node1's neighbors and node2's neighbors to the new
candidate pairs
                    update E1, E2, and EA
                    update N1, N2, and M for all other candidate pairs

                    done = False
                    break
                else
                    add pair to new_candidatePairs

        extend the candidatePairs with new_candidatePairs

    return (g1alignednodes, g2alignednodes, aligned_pairs)
```

Results:

Currently algorithm is too slow to run for any other pair besides Rat-Mouse, or Rat-Human. It is showing promising results, by finding perfectly aligned subgraphs of size ~30 for rat-mouse and size ~40 rat-human.The algorithm needs to be altered in order to produce alignments for other pairs in a reasonable amount of time.
The alignments are located in /extra/wayne0/preserve/utsavj/SANA/dijkstra/alignments.
The seed pairs are located at /home/hanweny/BLANT/alignment_seeds_new/.

Discussion:

Throughout the quarter the seed were produced using the Maximizer function. The Maximizer function looks for the k-node graphlet with largest canonical integer representation in a W-node window. In order to find graphlets with higher number of edges, the algorithm now uses DMAX to produce seeds. Maximizer with Distance (DMAX): the k-node graphlet has closest number of edges to the median (k choose 2) / 2. If there is a tie, choose the one with the largest canonical integer representation. In addition, another command line argument that should be added is the lower bound for edge density.

In order to speed up the processing time a skip list must be used to dynamically insert items and update the list. Currently, candidate pairs are being added to a list. It takes O(nlogn) time to reconstruct the list and sort it after aligning a pair. However, instead of updating edge_freq (hashmap that stores each pairs n1, n2, and M values) and sorting the whole candidate pairs again, it will be must faster to only remove the pairs that have to be updated in edge_freq from a skip list and reinsert them. This will cost only O(plogn) time where p is the number of pairs that have to be updated.